REAL TIME KERNEL

FreeRTOS™

FreeRTOS™ Gerçek Zamanlı Çekirdeğinde

UZMANLAŞMA

Uygulamalı Eğitim Kılavuzu

Mastering the FreeRTOS™ Real Time Kernel — A Hands-On Tutorial Guide

TÜRKÇE ÇEVİRİ | 2026

ORİJİNAL YAZAR

Richard Barry

The FreeRTOS Ekibi | Amazon Web Services

ÇEVİREN

Ertan Suluağaç

Kıdemli Gömülü Yazılım Uzmanı

Telif Hakkı ve İzin Bilgisi

Richard Barry ve The FreeRTOS Ekibi — Çeviren: Ertan Suluağaç

Orijinal Eser

Başlık: FreeRTOS™ Gerçek Zamanlı Çekirdeğinde Uzmanlaşma — Uygulamalı Eğitim Kılavuzu
(Mastering the FreeRTOS™ Real Time Kernel — A Hands-On Tutorial Guide)

Yazar: Richard Barry

Kaynak: https://github.com/FreeRTOS/FreeRTOS-Kernel-Book

Lisans — Creative Commons CC BY-SA 4.0

Orijinal eser **Creative Commons Attribution-ShareAlike 4.0 International Public License (CC BY-SA 4.0)** altında lisanslanmıştır.

Lisans Metni: https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/blob/main/LICENSE

Lisans Koşullarına Uyum

Atıf (Attribution — Bölüm 3a):

- **Yaratıcı:** Richard Barry ve The FreeRTOS Ekibi
- **Telif Hakkı:** © Richard Barry. Tüm hakları saklıdır.
- **Lisans:** CC BY-SA 4.0 International Public License
- **Garanti Reddi:** Lisansör materyali olduğu gibi sunar, hiçbir garanti vermez (Bölüm 5).
- **Değişiklik:** İngilizce'den Türkçe'ye çevrilmiştir.

Aynı Lisansla Paylaş (ShareAlike — Bölüm 3b):

- Bu çeviri de aynı **CC BY-SA 4.0** lisansı altında paylaşılmaktadır.
- Lisans: https://creativecommons.org/licenses/by-sa/4.0/
- Ek kısıtlama uygulanmamıştır.

Richard Barry'nin Onay E-postası

Gönderen: Richard Barry <r.barry@freertos.org>

"Apologies for the delay, I needed to check a few things.

It turns out the book contents are available under the Creative Commons Attribution-ShareAlike 4.0 International Public License which itself grants permission for your translation provided you comply with a few items (for example, retain the original creator identification, include a copyright notice, etc.). See https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/blob/main/LICENSE

Hope this makes things simple for you.

Regards,
Richard.”

© Orijinal eser: Richard Barry ve The FreeRTOS Ekibi.

© Türkçe çeviri: Ertan Suluağaç, 2026.

Orijinal eser ve bu çeviri CC BY-SA 4.0 lisansı altında lisanslanmıştır.

FreeRTOS ve FreeRTOS logosu, Amazon Web Services, Inc. veya bağlı kuruluşlarının ticari markalarıdır.

İÇİNDEKİLER

Bölüm 1 – Önsöz

1.1 Küçük Gömülü Sistemlerde Çoklu Görev
1.1.1 FreeRTOS Çekirdeği Hakkında
1.1.2 Değer Önerisi
1.1.3 Terminoloji Hakkında Bir Not
1.1.4 Neden Bir RTOS Kullanmalı?
1.1.5 FreeRTOS Çekirdek Özellikleri
1.1.6 Lisanslama ve FreeRTOS, OpenRTOS, SafeRTOS Ailesi
1.2 Dahil Edilen Kaynak Dosyaları ve Projeler
1.2.1 Bu Kitaba Eşlik Eden Örneklerin Elde Edilmesi

Bölüm 2 – FreeRTOS Çekirdek Dağıtımı

2.1 Giriş
2.2 FreeRTOS Dağıtımını Anlamak
2.2.1 Tanım: FreeRTOS Portu
2.2.2 FreeRTOS'u Derlemek
2.2.3 FreeRTOSConfig.h
2.2.4 Resmi Dağıtımlar
2.2.6 Tüm Portlarda Ortak FreeRTOS Kaynak Dosyaları
2.2.7 Bir Porta Özel FreeRTOS Kaynak Dosyaları
2.2.8 Include Yolları
2.2.9 Başlık Dosyaları
2.3 Demo Uygulamalar
2.4 FreeRTOS Projesi Oluşturma
2.4.1 Sağlanan Demo Projelerinden Birini Uyarlama
2.4.2 Sıfırdan Yeni Bir Proje Oluşturma
2.5 Veri Türleri ve Kodlama Stili Kılavuzu
2.5.1 Veri Türleri
2.5.2 Değişken Adları
2.5.3 Fonksiyon Adları
2.5.4 Biçimlendirme
2.5.5 Makro Adları
2.5.6 Aşırı Tür Dönüştürmenin Gerekçesi

Bölüm 3 – Yığın Bellek Yönetimi

3.1 Giriş
3.1.1 Ön Koşullar
3.1.2 Kapsam
3.1.3 Statik ve Dinamik Bellek Tahsisi Arasında Geçiş
3.1.4 Dinamik Bellek Tahsisi Kullanma
3.1.5 Dinamik Bellek Tahsisi Seçenekleri

3.2 Örnek Bellek Tahsis Şemaları
3.2.1 Heap_1
3.2.2 Heap_2
3.2.3 Heap_3
3.2.4 Heap_4
3.2.5 Heap_5
3.2.6 heap_5'i Başlatma: vPortDefineHeapRegions() API Fonksiyonu
3.3 Yığın İle İlgili Yardımcı Fonksiyonlar ve Makrolar
3.3.1 Yığın Başlangıç Adresini Tanımlama
3.3.2 xPortGetFreeHeapSize() API Fonksiyonu
3.3.3 xPortGetMinimumEverFreeHeapSize() API Fonksiyonu
3.3.4 vPortGetHeapStats() API Fonksiyonu
3.3.5 Görev Başına Yığın Kullanım İstatistiklerini Toplama
3.3.6 Malloc Başarısız Kanca Fonksiyonları
3.3.7 Görev Yığınlarını Hızlı Belleğe Yerleştirme
3.4 Statik Bellek Tahsisi Kullanma
3.4.1 Statik Bellek Tahsisini Etkinleştirme
3.4.2 Statik Dahili Çekirdek Belleği
3.4.2.1 vApplicationGetTimerTaskMemory
3.4.2.2 vApplicationGetIdleTaskMemory

Bölüm 4 – Görev Yönetimi
4.1 Giriş
4.1.1 Kapsam
4.2 Görev Fonksiyonları
4.3 Üst Düzey Görev Durumları
4.4 Görev Oluşturma
4.4.1 xTaskCreate() API Fonksiyonu
4.5 Görev Öncelikleri
4.5.1 Genel Çizgeleyici
4.5.2 Mimari İçin Optimize Edilmiş Çizgeleyici
4.6 Zaman Ölçümü ve Tick Kesmesi
4.7 Çalışmıyor Durumunun Genişletilmesi
4.7.1 Engellenmiş (Blocked) Durumu
4.7.2 Askıya Alınmış (Suspended) Durumu
4.7.3 Hazır (Ready) Durumu
4.7.4 Durum Geçiş Diyagramının Tamamlanması
4.7.5 vTaskDelayUntil() API Fonksiyonu
4.8 Boşta Görevi ve Boşta Görev Kanca Fonksiyonu
4.8.1 Boşta Görev Kanca Fonksiyonları
4.8.2 Boşta Görev Kanca Fonksiyonları Uygulamasındaki Kısıtlamalar
4.9 Bir Görevin Önceliğini Değiştirme
4.9.1 vTaskPrioritySet() API Fonksiyonu
4.9.2 uxTaskPriorityGet() API Fonksiyonu

4.10 Bir Görevi Silme
4.10.1 vTaskDelete() API Fonksiyonu
4.11 Thread Yerel Depolama ve Yeniden Girilebilirlik
4.11.1 C Çalışma Zamanı Thread Yerel Depolama Uygulamaları
4.11.2 Özel C Çalışma Zamanı Thread Yerel Depolama
4.11.3 Uygulama Thread Yerel Depolama
4.12 Çizgeleme Algoritmaları
4.12.1 Görev Durumları ve Olayların Özeti
4.12.2 Çizgeleme Algoritmasını Seçme
4.12.3 Zaman Dilimli Öncelikli Önleyici Çizgeleme
4.12.4 Zaman Dilimsiz Öncelikli Önleyici Çizgeleme
4.12.5 İşbirlikçi Çizgeleme

Bölüm 5 – Kuyruk Yönetimi
5.1 Giriş
5.1.1 Kapsam
5.2 Kuyruğun Özellikleri
5.2.1 Veri Depolama
5.2.2 Birden Fazla Görev Tarafından Erişim
5.2.3 Kuyruk Okumalarında Engelleme
5.2.4 Kuyruk Yazmalarında Engelleme
5.2.5 Birden Fazla Kuyrukta Engelleme
5.2.6 Kuyruk Oluşturma: Statik ve Dinamik Tahsisli Kuyruklar
5.3 Kuyruk Kullanma
5.3.1 xQueueCreate() API Fonksiyonu
5.3.2 xQueueSendToBack() ve xQueueSendToFront() API Fonksiyonları
5.3.3 xQueueReceive() API Fonksiyonu
5.3.4 uxQueueMessagesWaiting() API Fonksiyonu
5.4 Birden Fazla Kaynaktan Veri Alma
5.5 Büyük veya Değişken Boyutlu Verilerle Çalışma
5.5.1 İşaretçileri Kuyruğa Alma
5.5.2 Farklı Tür ve Uzunluklarda Veri Göndermek İçin Kuyruk Kullanma
5.6 Birden Fazla Kuyruktan Alma
5.6.1 Kuyruk Kümeleri
5.6.2 xQueueCreateSet() API Fonksiyonu
5.6.3 xQueueAddToSet() API Fonksiyonu
5.6.4 xQueueSelectFromSet() API Fonksiyonu
5.6.5 Daha Gerçekçi Kuyruk Kümesi Kullanım Senaryoları
5.7 Posta Kutusu Oluşturmak İçin Kuyruk Kullanma
5.7.1 xQueueOverwrite() API Fonksiyonu
5.7.2 xQueuePeek() API Fonksiyonu

Bölüm 6 – Yazılım Zamanlayıcı Yönetimi
6.1 Bölüm Girişi ve Kapsamı

6.1.1 Kapsam
6.2 Yazılım Zamanlayıcı Geri Çağırma Fonksiyonları
6.3 Yazılım Zamanlayıcısının Özellikleri ve Durumları
6.3.1 Yazılım Zamanlayıcısının Periyodu
6.3.2 Tek Seferlik ve Otomatik Yeniden Yükleme Zamanlayıcıları
6.3.3 Yazılım Zamanlayıcı Durumları
6.4 Yazılım Zamanlayıcısının Bağlamı
6.4.1 RTOS Daemon (Zamanlayıcı Hizmeti) Görevi
6.4.2 Zamanlayıcı Komut Kuyruğu
6.4.3 Daemon Görev Çizgeleme
6.5 Yazılım Zamanlayıcısı Oluşturma ve Başlatma
6.5.1 xTimerCreate() API Fonksiyonu
6.5.2 xTimerStart() API Fonksiyonu
6.6 Zamanlayıcı Kimliği (Timer ID)
6.6.1 vTimerSetTimerID() API Fonksiyonu
6.6.2 pvTimerGetTimerID() API Fonksiyonu
6.7 Zamanlayıcı Periyodunu Değiştirme
6.7.1 xTimerChangePeriod() API Fonksiyonu
6.8 Yazılım Zamanlayıcısını Sıfırlama
6.8.1 xTimerReset() API Fonksiyonu

Bölüm 7 – Kesme Yönetimi

7.1 Giriş
7.1.1 Olaylar
7.1.2 Kapsam
7.2 Bir ISR İçinden FreeRTOS API'sini Kullanma
7.2.1 Kesme Güvenli API
7.2.2 Ayrı Bir Kesme Güvenli API Kullanmanın Faydaları
7.2.3 Ayrı Bir Kesme Güvenli API Kullanmanın Dezavantajları
7.2.4 xHigherPriorityTaskWoken Parametresi
7.2.5 portYIELD_FROM_ISR() ve portEND_SWITCHING_ISR() Makroları
7.3 Ertelenmiş Kesme İşleme
7.4 Senkronizasyon İçin Kullanılan İkili Semaforlar
7.4.1 xSemaphoreCreateBinary() API Fonksiyonu
7.4.2 xSemaphoreTake() API Fonksiyonu
7.4.3 xSemaphoreGiveFromISR() API Fonksiyonu
7.4.4 Örnek 7.1'de Kullanılan Görevin Geliştirilmesi
7.5 Sayıcı Semaforlar
7.5.1 xSemaphoreCreateCounting() API Fonksiyonu
7.6 RTOS Daemon Görevine İş Erteleme
7.6.1 xTimerPendFunctionCallFromISR() API Fonksiyonu
7.7 Kesme Hizmet Rutini İçinde Kuyruk Kullanma
7.7.1 xQueueSendToFrontFromISR() ve xQueueSendToBackFromISR() API Fonksiyonları
7.7.2 Bir ISR'den Kuyruk Kullanırken Dikkat Edilecekler

7.8 Kesme İç İçe Geçme
7.8.1 ARM Cortex-M ve ARM GIC Kullanıcıları İçin Not

Bölüm 8 – Kaynak Yönetimi

8.1 Bölüm Girişi ve Kapsamı
8.1.1 Karşılıklı Dışlama
8.1.2 Kapsam
8.2 Kritik Bölümler ve Çizgeleyiciyi Askıya Alma
8.2.1 Temel Kritik Bölümler
8.2.2 Çizgeleyiciyi Askıya Alma (veya Kilitleme)
8.2.3 vTaskSuspendAll() API Fonksiyonu
8.2.4 xTaskResumeAll() API Fonksiyonu
8.3 Mutex'ler (ve İkili Semaforlar)
8.3.1 xSemaphoreCreateMutex() API Fonksiyonu
8.3.2 Öncelik Terslemesi
8.3.3 Öncelik Devralma
8.3.4 Ölümcül Kilitlenme (Deadlock)
8.3.5 Özyinelemeli Mutex'ler
8.3.6 Mutex'ler ve Görev Çizgeleme
8.4 Kapıcı (Gatekeeper) Görevleri
8.4.1 vPrintString() Fonksiyonunu Kapıcı Görev Kullanarak Yeniden Yazma

Bölüm 9 – Olay Grupları

9.1 Bölüm Girişi ve Kapsamı
9.1.1 Kapsam
9.2 Olay Grubunun Özellikleri
9.2.1 Olay Grupları, Olay Bayrakları ve Olay Bitleri
9.2.2 EventBits_t Veri Türü Hakkında
9.2.3 Birden Fazla Görev Tarafından Erişim
9.2.4 Olay Grubu Kullanmanın Pratik Bir Örneği
9.3 Olay Grupları Kullanarak Olay Yönetimi
9.3.1 xEventGroupCreate() API Fonksiyonu
9.3.2 xEventGroupSetBits() API Fonksiyonu
9.3.3 xEventGroupSetBitsFromISR() API Fonksiyonu
9.3.4 xEventGroupWaitBits() API Fonksiyonu
9.3.5 xEventGroupGetStaticBuffer() API Fonksiyonu
9.4 Olay Grubu Kullanarak Görev Senkronizasyonu
9.4.1 xEventGroupSync() API Fonksiyonu

Bölüm 10 – Görev Bildirimleri

10.1 Giriş
10.1.1 Aracı Nesneler Üzerinden İletişim
10.1.2 Görev Bildirimleri — Göreve Doğrudan İletişim
10.1.3 Kapsam

10.2 Görev Bildirimleri; Faydaları ve Kısıtlamaları
10.2.1 Görev Bildirimlerinin Performans Avantajları
10.2.2 Görev Bildirimlerinin RAM Alanı Avantajları
10.2.3 Görev Bildirimlerinin Kısıtlamaları
10.3 Görev Bildirimlerini Kullanma
10.3.1 Görev Bildirimi API Seçenekleri
10.3.1.1 API Fonksiyonlarının Tam Listesi
10.3.2 xTaskNotifyGive() API Fonksiyonları
10.3.3 vTaskNotifyGiveFromISR() API Fonksiyonu
10.3.4 ulTaskNotifyTake() API Fonksiyonu
10.3.5 xTaskNotify() ve xTaskNotifyFromISR() API Fonksiyonları
10.3.6 xTaskNotifyWait() API Fonksiyonu
10.3.7 Çevre Birimi Sürücülerinde Görev Bildirimleri: UART Örneği
10.3.8 Çevre Birimi Sürücülerinde Görev Bildirimleri: ADC Örneği
10.3.9 Bir Uygulama İçinde Doğrudan Görev Bildirimi Kullanma

Bölüm 11 – Düşük Güç Desteği
11.1 Güç Tasarrufu Girişi
11.2 FreeRTOS Uyku Modları
11.3 Yerleşik Tick'siz Boşta İşlevselliğini Etkinleştirme Fonksiyonları
11.3.1 portSUPPRESS_TICKS_AND_SLEEP() Makrosu
11.3.2 vPortSuppressTicksAndSleep Fonksiyonu
11.3.3 eTaskConfirmSleepModeStatus Fonksiyonu
11.3.4 configPRE_SLEEP_PROCESSING Yapılandırması
11.3.5 configPOST_SLEEP_PROCESSING Yapılandırması
11.4 portSUPPRESS_TICKS_AND_SLEEP() Makrosunun Uygulanması
11.5 Boşta Görev Kanca Fonksiyonu

Bölüm 12 – Geliştirici Desteği
12.1 Giriş
12.2 configASSERT()
12.2.1 Örnek configASSERT() Tanımları
12.3 FreeRTOS için Tracealyzer
12.4 Hata Ayıklama İle İlgili Kanca (Geri Çağırma) Fonksiyonları
12.4.1 Malloc Başarısız Kancası
12.4.2 Yığın Taşması Kancası
12.5 Çalışma Zamanı ve Görev Durum Bilgilerini Görüntüleme
12.5.1 Görev Çalışma Zamanı İstatistikleri
12.5.2 Çalışma Zamanı İstatistik Saati
12.5.3 Çalışma Zamanı İstatistiklerini Toplamak İçin Uygulamayı Yapılandırma
12.5.4 uxTaskGetSystemState() API Fonksiyonu
12.5.5 vTaskListTasks() Yardımcı Fonksiyonu
12.5.6 vTaskGetRunTimeStatistics() Yardımcı Fonksiyonu
12.5.7 Çalışma Zamanı İstatistikleri Oluşturma ve Görüntüleme Örneği

12.6 İzleme Kanca Makroları
12.6.1 Kullanılabilir İzleme Kanca Makroları
12.6.2 İzleme Kanca Makrolarını Tanımlama
12.6.3 FreeRTOS Uyumlu Hata Ayıklayıcı Eklentileri

Bölüm 13 – Sorun Giderme
13.1 Bölüm Girişi ve Kapsamı
13.2 Kesme Öncelikleri
13.3 Yığın Taşması
13.3.1 uxTaskGetStackHighWaterMark() API Fonksiyonu
13.3.2 Çalışma Zamanı Yığın Kontrolü — Genel Bakış
13.3.3 Çalışma Zamanı Yığın Kontrolü — Yöntem 1
13.3.4 Çalışma Zamanı Yığın Kontrolü — Yöntem 2
13.3.5 Çalışma Zamanı Yığın Kontrolü — Yöntem 3
13.4 printf() ve sprintf() Kullanımı
13.4.1 Printf-stdarg.c
13.5 Diğer Yaygın Hata Kaynakları
13.5.1 Belirti: Bir demo'ya basit bir görev eklemek demo'nun çökmesine neden oluyor
13.5.2 Belirti: Bir kesme içinde API fonksiyonu kullanmak uygulamanın çökmesine neden oluyor
13.5.3 Belirti: Bazen uygulama bir kesme hizmet rutini içinde çöküyor
13.5.4 Belirti: İlk görevi başlatmaya çalışırken çizgeleyici çöküyor
13.5.5 Belirti: Kesmeler beklenmedik şekilde devre dışı bırakılıyor veya kritik bölümler düzgün iç içe geçmiyor
13.5.6 Belirti: Çizgeleyici başlamadan önce bile uygulama çöküyor
13.5.7 Belirti: Çizgeleyici askıya alınmışken veya kritik bölüm içinde API fonksiyonu çağırmak uygulamanın çökmesine neden oluyor
13.6 Ek Hata Ayıklama Adımları

Örnekler Listesi

Örnek 4.1 — Görev Oluşturma
Örnek 4.2 — Görev Parametresini Kullanma
Örnek 4.3 — Önceliklerle Denemeler
Örnek 4.4 — Gecikme Oluşturmak İçin Engellenmiş Durumu Kullanma
Örnek 4.5 — Görevleri vTaskDelayUntil() Kullanacak Şekilde Dönüştürme
Örnek 4.6 — Engelleyen ve Engellemyen Görevleri Birleştirme
Örnek 4.7 — Boşta Görev Kanca Fonksiyonu Tanımlama
Örnek 4.8 — Görev Önceliklerini Değiştirme
Örnek 4.9 — Görevleri Silme
Örnek 5.1 — Kuyruktan Alırken Engelleme
Örnek 5.2 — Kuyruğa Gönderirken Engelleme ve Kuyrukta Yapı Gönderme
Örnek 5.3 — Kuyruk Kümesi Kullanma
Örnek 6.1 — Tek Seferlik ve Otomatik Yeniden Yükleme Zamanlayıcıları Oluşturma
Örnek 6.2 — Geri Çağırma Fonksiyonu Parametresini ve Zamanlayıcı Kimliğini Kullanma

Örnek 6.3 — Yazılım Zamanlayıcısını Sıfırlama
Örnek 7.1 — Bir Görevi Kesme İle Senkronize Etmek İçin İkili Semafor Kullanma
Örnek 7.2 — Bir Görevi Kesme İle Senkronize Etmek İçin Sayıcı Semafor Kullanma
Örnek 7.3 — Merkezi Ertelenmiş Kesme İşleme
Örnek 7.4 — Kesme İçinde Kuyrukta Gönderme ve Alma
Örnek 8.1 — vPrintString() Fonksiyonunu Semafor Kullanarak Yeniden Yazma
Örnek 8.2 — Yazdırma Görevi İçin Alternatif Uygulama
Örnek 9.1 — Olay Gruplarıyla Denemeler
Örnek 9.2 — Görevleri Senkronize Etme
Örnek 10.1 — Semafor Yerine Görev Bildirimi Kullanma, Yöntem 1
Örnek 10.2 — Semafor Yerine Görev Bildirimi Kullanma, Yöntem 2

Şekiller Listesi

Şekil 2.1 — FreeRTOS dağıtımındaki üst düzey dizinler
Şekil 2.2 — FreeRTOS dizin ağacındaki çekirdek kaynak dosyaları
Şekil 2.3 — FreeRTOS dizin ağacındaki porta özel kaynak dosyaları
Şekil 2.4 — Demo dizin hiyerarşisi
Şekil 3.1 — Heap_1 dizisinden her görev oluşturulduğunda RAM tahsisi
Şekil 3.2 — Görevler oluşturulup silinirken heap_2 dizisinden RAM tahsis ve serbest bırakma
Şekil 3.3 — Heap_4 dizisinden RAM tahsis ve serbest bırakma
Şekil 3.4 — Bellek Haritası
Şekil 4.1 — Üst düzey görev durumları ve geçişleri
Şekil 4.2 — Örnek 4.1 yürütüldüğünde üretilen çıktı
Şekil 4.3 — İki Örnek 4.1 görevinin gerçek yürütme deseni
Şekil 4.4 — Tick kesmesinin yürütülmesini göstermek için genişletilmiş yürütme dizisi
Şekil 4.5 — Her iki görevi farklı önceliklerle çalıştırma
Şekil 4.6 — Bir görev diğerinden daha yüksek önceliğe sahip olduğunda yürütme deseni
Şekil 4.7 — Tam görev durum makinesi
Şekil 4.8 — Örnek 4.4 yürütüldüğünde üretilen çıktı
Şekil 4.9 — vTaskDelay() kullanıldığında yürütme dizisi
Şekil 4.10 — Kalın çizgiler görevler tarafından gerçekleştirilen durum geçişlerini gösterir
Şekil 4.11 — Örnek 4.6 yürütüldüğünde üretilen çıktı
Şekil 4.12 — Örnek 4.6 yürütme deseni
Şekil 4.13 — Örnek 4.7 yürütüldüğünde üretilen çıktı
Şekil 4.14 — Örnek 4.8 çalıştırılırken görev yürütme dizisi
Şekil 4.15 — Örnek 4.8 yürütüldüğünde üretilen çıktı
Şekil 4.16 — Örnek 4.9 yürütüldüğünde üretilen çıktı
Şekil 4.17 — Örnek 4.9 için yürütme dizisi
Şekil 4.18 — Görev önceliklendirme ve ön almayı vurgulayan yürütme deseni
Şekil 4.19 — Görev önceliklendirme ve zaman dilimlemeyi vurgulayan yürütme deseni
Şekil 4.20 — Şekil 4.19 ile aynı senaryo için yürütme deseni
Şekil 4.21 — Eşit öncelikli görevlerin nasıl çalışabileceğini gösteren yürütme deseni
Şekil 4.22 — İşbirlikçi çizgeleyicinin davranışını gösteren yürütme deseni
Şekil 5.1 — Kuyrukta yazma ve okuma örnek dizisi
Şekil 5.2 — Örnek 5.1 yürütüldüğünde üretilen çıktı
Şekil 5.3 — Örnek 5.1 tarafından üretilen yürütme dizisi
Şekil 5.4 — Kuyrukta yapı gönderme senaryosu
Şekil 5.5 — Örnek 5.2 tarafından üretilen çıktı
Şekil 5.6 — Örnek 5.2 tarafından üretilen yürütme dizisi
Şekil 5.7 — Örnek 5.3 yürütüldüğünde üretilen çıktı

Şekil 6.1 — Tek seferlik ve otomatik yeniden yükleme zamanlayıcıları arasındaki davranış farkı
Şekil 6.2 — Otomatik yeniden yükleme zamanlayıcı durumları ve geçişleri
Şekil 6.3 — Tek seferlik zamanlayıcı durumları ve geçişleri
Şekil 6.4 — Zamanlayıcı komut kuyruğunun RTOS daemon görevi ile iletişimi
Şekil 6.5 — xTimerStart() çağıran görevin önceliği daemon görevinden yüksek olduğunda yürütme deseni
Şekil 6.6 — xTimerStart() çağıran görevin önceliği daemon görevinden düşük olduğunda yürütme deseni
Şekil 6.7 — Örnek 6.1 yürütüldüğünde üretilen çıktı
Şekil 6.8 — Örnek 6.2 yürütüldüğünde üretilen çıktı
Şekil 6.9 — 6 tick periyodlu bir yazılım zamanlayıcısını başlatma ve sıfırlama
Şekil 6.10 — Örnek 6.3 yürütüldüğünde üretilen çıktı
Şekil 7.1 — Yüksek öncelikli görevde kesme işlemenin tamamlanması
Şekil 7.2 — Ertelenmiş kesme işleme için ikili semafor kullanımı
Şekil 7.3 — Bir görevi kesme ile senkronize etmek için ikili semafor kullanımı
Şekil 7.4 — Örnek 7.1 yürütüldüğünde üretilen çıktı
Şekil 7.5 — Örnek 7.1 yürütüldüğündeki yürütme dizisi
Şekil 7.6 — Görev ilk olayı işlemeden önce bir kesme gerçekleştiğinde senaryo
Şekil 7.7 — Görev ilk olayı işlemeden önce iki kesme gerçekleştiğinde senaryo
Şekil 7.8 — Sayıcı semafor kullanımı
Şekil 7.9 — Örnek 7.2 yürütüldüğünde üretilen çıktı
Şekil 7.10 — Örnek 7.3 yürütüldüğünde üretilen çıktı
Şekil 7.11 — Örnek 7.3 yürütüldüğündeki yürütme dizisi
Şekil 7.12 — Örnek 7.4 yürütüldüğünde üretilen çıktı
Şekil 7.13 — Örnek 7.4 tarafından üretilen yürütme dizisi
Şekil 7.14 — Kesme iç içe geçme davranışını etkileyen sabitler
Şekil 7.15 — Cortex-M mikrodenetleyicisinde ikili 101 önceliğinin saklanması
Şekil 8.1 — Mutex kullanılarak uygulanan karşılıklı dışlama
Şekil 8.2 — Örnek 8.1 yürütüldüğünde üretilen çıktı
Şekil 8.3 — Örnek 8.1 için olası bir yürütme dizisi
Şekil 8.4 — En kötü durum öncelik terslenme senaryosu
Şekil 8.5 — Öncelik devralma ile öncelik terslenme etkisini en aza indirme
Şekil 8.6 — Aynı önceliğe sahip görevlerin aynı mutex kullandığı olası yürütme dizisi
Şekil 8.7 — Aynı öncelikte iki görev oluşturulduğunda gerçekleşebilecek yürütme dizisi
Şekil 8.8 — Örnek 8.2 yürütüldüğünde üretilen çıktı
Şekil 9.1 — EventBits_t türündeki değişkende olay bayrak-bit eşlemesi
Şekil 9.2 — Yalnızca bit 1, 4 ve 7 ayarlanmış bir olay grubu
Şekil 9.3 — xWaitForAllBits pdFALSE olarak ayarlandığında Örnek 9.1 çıktısı
Şekil 9.4 — xWaitForAllBits pdTRUE olarak ayarlandığında Örnek 9.1 çıktısı
Şekil 9.5 — Örnek 9.2 yürütüldüğünde üretilen çıktı
Şekil 10.1 — Görevler arası olay gönderimi için iletişim nesnesi kullanımı
Şekil 10.2 — Görevler arası doğrudan olay gönderimi için görev bildirimi kullanımı
Şekil 10.3 — Örnek 10.1 yürütüldüğünde üretilen çıktı
Şekil 10.4 — Örnek 10.1 yürütüldüğündeki yürütme dizisi
Şekil 10.5 — Örnek 10.2 yürütüldüğünde üretilen çıktı
Şekil 10.6 — Uygulama görevlerinden bulut sunucusuna ve geri iletişim yolları
Şekil 12.1 — FreeRTOS+Trace 20'den fazla birbirine bağlı görünüm içerir
Şekil 12.2 — FreeRTOS+Trace ana izleme görünümü
Şekil 12.3 — FreeRTOS+Trace CPU yük görünümü
Şekil 12.4 — FreeRTOS+Trace yanıt süresi görünümü
Şekil 12.5 — FreeRTOS+Trace kullanıcı olay çizim görünümü
Şekil 12.6 — FreeRTOS+Trace çekirdek nesne geçmişi görünümü

Şekil 12.7 — vTaskListTasks() tarafından üretilen örnek çıktı
Şekil 12.8 — vTaskGetRunTimeStatistics() tarafından üretilen örnek çıktı

Kod Listeleri

Liste 2.1 — Yeni bir main() işlevi için şablon
Liste 3.1 — vPortDefineHeapRegions() API işlev prototipi
Liste 3.2 — HeapRegion_t yapısı
Liste 3.3 — RAM bölgelerinin tamamını tanımlayan HeapRegion_t dizisi
Liste 3.4 — RAM2 ve RAM3'ün tamamını ama RAM1'in yalnızca bir bölümünü tanımlayan HeapRegion_t dizisi
Liste 3.5 — GCC sözdizimi ile heap_4 dizisini bildirme ve .my_heap bölümüne yerleştirme
Liste 3.6 — IAR sözdizimi ile heap_4 dizisini bildirme ve 0x20000000 adresine yerleştirme
Liste 3.7 — xPortGetFreeHeapSize() API işlev prototipi
Liste 3.8 — xPortGetMinimumEverFreeHeapSize() API işlev prototipi
Liste 3.9 — vPortGetHeapStats() API işlev prototipi
Liste 3.10 — HeapStats_t yapısı
Liste 3.11 — Malloc başarısız kanca işlev adı ve prototipi
Liste 3.12 — pvPortMallocStack() ve vPortFreeStack() makrolarını uygulama tanımlı bellek tahsisine eşleme
Liste 3.13 — vApplicationGetTimerTaskMemory tipik uygulaması
Liste 3.14 — vApplicationGetIdleTaskMemory tipik uygulaması
Liste 4.1 — Görev fonksiyon prototipi
Liste 4.2 — Tipik bir görev fonksiyonunun yapısı
Liste 4.3 — xTaskCreate() API işlev prototipi
Liste 4.4 — Örnek 4.1'de kullanılan birinci görevin uygulanması
Liste 4.5 — Örnek 4.1'de kullanılan ikinci görevin uygulanması
Liste 4.6 — Örnek 4.1 görevlerinin başlatılması
Liste 4.7 — Çizgeleyici başlatıldıktan sonra bir görev içinden başka görev oluşturma
Liste 4.8 — Örnek 4.2'de iki görev oluşturmak için kullanılan tek görev fonksiyonu
Liste 4.9 — Örnek 4.2 için main() fonksiyonu
Liste 4.10 — pdMS_TO_TICKS() makrosunu kullanarak 200 milisaniyeyi tick'e dönüştürme
Liste 4.11 — Farklı önceliklerde iki görev oluşturma
Liste 4.12 — vTaskDelay() API işlev prototipi
Liste 4.13 — Null döngü gecikmesi yerine vTaskDelay() çağrısı kullanan görev
Liste 4.14 — vTaskDelayUntil() API işlev prototipi
Liste 4.15 — vTaskDelayUntil() kullanan görev uygulaması
Liste 4.16 — Örnek 4.6'da kullanılan sürekli işleme görevi
Liste 4.17 — Örnek 4.6'da kullanılan periyodik görev
Liste 4.18 — Boşta görev kanca fonksiyon adı ve prototipi
Liste 4.19 — Çok basit bir boşta kanca fonksiyonu
Liste 4.20 — ulIdleCycleCount değerini yazdıran görev
Liste 4.21 — vTaskPrioritySet() API işlev prototipi
Liste 4.22 — uxTaskPriorityGet() API işlev prototipi
Liste 4.23 — Örnek 4.8'de Görev 1'in uygulaması
Liste 4.24 — Örnek 4.8'de Görev 2'nin uygulaması
Liste 4.25 — Örnek 4.8 için main() uygulaması

Liste 4.26 — vTaskDelete() API işlev prototipi
Liste 4.27 — Örnek 4.9 için main() uygulaması
Liste 4.28 — Örnek 4.9 için Görev 1'in uygulaması
Liste 4.29 — Örnek 4.9 için Görev 2'nin uygulaması
Liste 4.30 — Thread Local Storage Pointer API fonksiyon prototipleri
Liste 5.1 — xQueueCreate() API işlev prototipi
Liste 5.2 — xQueueSendToFront() API işlev prototipi
Liste 5.3 — xQueueSendToBack() API işlev prototipi
Liste 5.4 — xQueueReceive() API işlev prototipi
Liste 5.5 — uxQueueMessagesWaiting() API işlev prototipi
Liste 5.6 — Örnek 5.1'de gönderme görevinin uygulaması
Liste 5.7 — Örnek 5.1 için alıcı görevin uygulaması
Liste 5.8 — Örnek 5.1'de main() uygulaması
Liste 5.9 — Kuyrukta gönderilecek yapının tanımı
Liste 5.10 — Örnek 5.2'de gönderme görevinin uygulaması
Liste 5.11 — Örnek 5.2 için alıcı görevin tanımı
Liste 5.12 — Örnek 5.2 için main() uygulaması
Liste 5.13 — İşaretçi tutan bir kuyruk oluşturma
Liste 5.14 — Bir kuyruğa tampon işaretçisi gönderme
Liste 5.15 — Bir kuyruktan tampon işaretçisi alma
Liste 5.16 — FreeRTOS+TCP'de TCP/IP yığın görevine olay göndermek için kullanılan yapı
Liste 5.17 — Ağdan alınan verileri TCP/IP görevine göndermek için IPStackEvent_t kullanımı
Liste 5.18 — Bağlantı kabul eden soketin tanıtıcısını TCP/IP görevine göndermek için IPStackEvent_t kullanımı
Liste 5.19 — TCP/IP görevine ağ kapatma olayı göndermek için IPStackEvent_t kullanımı
Liste 5.20 — IPStackEvent_t yapısının alınması ve işlenmesi
Liste 5.21 — xQueueCreateSet() API işlev prototipi
Liste 5.22 — xQueueAddToSet() API işlev prototipi
Liste 5.23 — xQueueSelectFromSet() API işlev prototipi
Liste 5.24 — Örnek 5.3 için main() uygulaması
Liste 5.25 — Örnek 5.3'te kullanılan gönderme görevleri
Liste 5.26 — Örnek 5.3'te kullanılan alma görevi
Liste 5.27 — Kuyruk ve semaforlar içeren bir kuyruk kümesi kullanma
Liste 5.28 — Posta kutusu olarak kullanılmak üzere bir kuyruk oluşturma
Liste 5.29 — xQueueOverwrite() API işlev prototipi
Liste 5.30 — xQueueOverwrite() API fonksiyonunun kullanımı
Liste 5.31 — xQueuePeek() API işlev prototipi
Liste 5.32 — xQueuePeek() API fonksiyonunun kullanımı
Liste 6.1 — Yazılım zamanlayıcı geri çağırma fonksiyon prototipi
Liste 6.2 — xTimerDelete() API işlev prototipi
Liste 6.3 — xTimerCreate() API işlev prototipi
Liste 6.4 — xTimerStart() API işlev prototipi
Liste 6.5 — Örnek 6.1'de tek seferlik ve otomatik yeniden yükleme zamanlayıcıları oluşturma
Liste 6.6 — Örnek 6.1'de kullanılan geri çağırma fonksiyonları
Liste 6.7 — vTimerSetTimerID() API işlev prototipi
Liste 6.8 — pvTimerGetTimerID() API işlev prototipi
Liste 6.9 — Örnek 6.2'de kullanılan geri çağırma fonksiyonu

Liste 6.10 — xTimerChangePeriod() API işlev prototipi
Liste 6.11 — xTimerReset() API işlev prototipi
Liste 6.12 — Örnek 6.3'ün main() fonksiyonu
Liste 6.13 — Örnek 6.3'te kullanılan geri çağırma fonksiyonu
Liste 6.14 — xTimerReset() API işlev prototipi
Liste 6.15 — Örnek 6.3'te tek seferlik zamanlayıcı için geri çağırma fonksiyonu
Liste 6.16 — Örnek 6.3'te yazılım zamanlayıcısını sıfırlayan görev
Liste 7.1 — xSemaphoreCreateBinary() API işlev prototipi
Liste 7.2 — xSemaphoreTake() API işlev prototipi
Liste 7.3 — xSemaphoreGiveFromISR() API işlev prototipi
Liste 7.4 — Örnek 7.1'de kullanılan görev uygulaması
Liste 7.5 — Örnek 7.1'de kullanılan yazılım kesmesi uygulaması
Liste 7.6 — Örnek 7.1'de kullanılan main() uygulaması
Liste 7.7 — Geliştirilmiş görev uygulaması (Örnek 7.1)
Liste 7.8 — xSemaphoreCreateCounting() API işlev prototipi
Liste 7.9 — Örnek 7.2'de kullanılan main() uygulaması
Liste 7.10 — xTimerPendFunctionCallFromISR() API işlev prototipi
Liste 7.11 — Örnek 7.3'te kullanılan yazılım kesmesi uygulaması
Liste 7.12 — Örnek 7.3'te daemon görevine ertelenen fonksiyon
Liste 7.13 — Örnek 7.3'te kullanılan main() uygulaması
Liste 7.14 — xQueueSendToFrontFromISR() API işlev prototipi
Liste 7.15 — xQueueSendToBackFromISR() API işlev prototipi
Liste 7.16 — Örnek 7.4'te veri kaynağı olarak kullanılan görev
Liste 7.17 — Örnek 7.4'te kullanılan kesme hizmet rutini
Liste 7.18 — Örnek 7.4'te kullanılan main() uygulaması
Liste 7.19 — xQueueSendToFrontFromISR() API işlev prototipi
Liste 7.20 — xQueueSendToBackFromISR() API işlev prototipi
Liste 7.21 — Örnek 7.4'te kuyruğa yazan görevin uygulaması
Liste 7.22 — Örnek 7.4'te kullanılan kesme hizmet rutini uygulaması
Liste 7.23 — Örnek 7.4'te ISR'dan alınan dizeleri yazdıran görev
Liste 7.24 — Örnek 7.4 için main() fonksiyonu
Liste 8.1 — taskENTER_CRITICAL() ve taskEXIT_CRITICAL() kullanım örneği
Liste 8.2 — taskENTER_CRITICAL_FROM_ISR() ve
taskEXIT_CRITICAL_FROM_ISR() kullanım örneği
Liste 8.3 — vTaskSuspendAll() API işlev prototipi
Liste 8.4 — xTaskResumeAll() API işlev prototipi
Liste 8.5 — Çizgeleyiciyi askıya alan vPrintString() uygulaması
Liste 8.6 — vPrintString() kullanan görev uygulaması
Liste 8.7 — Çizgeleyici kilitleme yöntemi kullanan Örnek 8.1 main() uygulaması
Liste 8.8 — xSemaphoreCreateMutex() API işlev prototipi
Liste 8.9 — Mutex ile korunan vPrintString() uygulaması
Liste 8.10 — Mutex kullanan prvNewPrintString() görev uygulaması
Liste 8.11 — Mutex yöntemi kullanan Örnek 8.1 main() uygulaması
Liste 8.12 — Ölümcül kilitlenme oluşturan senaryo
Liste 8.13 — Örnek 8.2 için main() uygulaması
Liste 8.14 — Kapıcı görev uygulaması
Liste 8.15 — Yazdırma görevleri uygulaması
Liste 8.16 — Tick kancası uygulaması
Liste 8.17 — Tick kanca fonksiyonu adı ve prototipi
Liste 8.18 — Kapı bekçisi görevi

Liste 8.19 — Örnek 8.2 için yazdırma görevi uygulaması
Liste 8.20 — Tick kanca uygulaması
Liste 8.21 — Örnek 8.2 için main() uygulaması
Liste 9.1 — xEventGroupCreate() API işlev prototipi
Liste 9.2 — xEventGroupSetBits() API işlev prototipi
Liste 9.3 — xEventGroupSetBitsFromISR() API işlev prototipi
Liste 9.4 — xEventGroupWaitBits() API işlev prototipi
Liste 9.5 — Olay grubu bitleri ayarlayan görevler (Örnek 9.1)
Liste 9.6 — Olay gruplarını bekleyen görev (Örnek 9.1)
Liste 9.7 — Örnek 9.1 için main() uygulaması
Liste 9.8 — xEventGroupGetStaticBuffer() API işlev prototipi
Liste 9.9 — xEventGroupSync() API işlev prototipi
Liste 9.10 — Örnek 9.2 için main() uygulaması
Liste 9.11 — Örnek 9.2'de kullanılan görev (Buluşma noktası)
Liste 9.12 — xEventGroupSync() API işlev prototipi
Liste 9.13 — Örnek 9.2'de kullanılan görevin uygulaması
Liste 9.14 — Örnek 9.2'de kullanılan main() fonksiyonu
Liste 10.1 — xTaskNotifyGive() API işlev prototipi
Liste 10.2 — vTaskNotifyGiveFromISR() API işlev prototipi
Liste 10.3 — ulTaskNotifyTake() API işlev prototipi
Liste 10.4 — Örnek 10.1'de kullanılan görev uygulaması
Liste 10.5 — Örnek 10.1'de kullanılan kesme hizmet rutini
Liste 10.6 — Örnek 10.2'de kullanılan görev uygulaması
Liste 10.7 — xTaskNotify() ve xTaskNotifyFromISR() API işlev prototipleri
Liste 10.8 — xTaskNotifyWait() API işlev prototipi
Liste 10.9 — UART alma görevi: İlk uygulama
Liste 10.10 — UART alma görevi: Optimize edilmiş uygulama
Liste 10.11 — ADC görevi uygulaması
Liste 10.12 — Görev bildirimi ile sürücü kütüphanesi alma fonksiyonu sözde kodu
Liste 10.13 — Görev bildirimi ile göreve değer aktarma sözde kodu
Liste 10.14 — Sunucu görevine kuyruğa gönderilen yapı ve veri türü
Liste 10.15 — Cloud Read API fonksiyonunun uygulaması
Liste 10.16 — Sunucu görevinin okuma isteğini işlemesi
Liste 10.17 — Cloud Write API fonksiyonunun uygulaması
Liste 10.18 — Sunucu görevinin gönderme isteğini işlemesi
Liste 11.1 — portSUPPRESS_TICKS_AND_SLEEP() makro prototipi
Liste 11.2 — vPortSuppressTicksAndSleep() işlev prototipi
Liste 11.3 — eTaskConfirmSleepModeStatus API işlev prototipi
Liste 11.4 — configPRE_SLEEP_PROCESSING makro prototipi
Liste 11.5 — configPOST_SLEEP_PROCESSING makro prototipi
Liste 11.6 — Kullanıcı tanımlı portSUPPRESS_TICKS_AND_SLEEP() uygulaması örneği
Liste 11.7 — vApplicationIdleHook API işlev prototipi
Liste 12.1 — configASSERT() tanımı: Hata ayıklayıcıda döngüye sokma
Liste 12.2 — configASSERT() tanımı: Dosya adı ve satır numarası kaydetme
Liste 12.3 — Onaylama başarısız olan kaynak kod satırını kaydeden configASSERT() tanımı
Liste 12.4 — uxTaskGetSystemState() API işlev prototipi
Liste 12.5 — TaskStatus_t yapısı
Liste 12.6 — vTaskListTasks() API işlev prototipi

Liste 12.7 — vTaskList() API işlev prototipi
Liste 12.8 — vTaskGetRunTimeStatistics() API işlev prototipi
Liste 12.9 — vTaskGetRunTimeStats() API işlev prototipi
Liste 12.10 — Zamanlayıcı taşmalarını saymak için 16-bit zamanlayıcı taşma kesme işleyicisi
Liste 12.11 — Çalışma zamanı istatistik toplamayı etkinleştirmek için FreeRTOSConfig.h'a eklenen makrolar
Liste 12.12 — Toplanan çalışma zamanı istatistiklerini yazdıran görev
Liste 13.1 — uxTaskGetStackHighWaterMark() API işlev prototipi
Liste 13.2 — uxTaskGetStackHighWaterMark2() API işlev prototipi
Liste 13.3 — Yığın taşması kanca fonksiyonu prototipi

Tablolar Listesi

Tablo 1 — Projeye dahil edilecek FreeRTOS kaynak dosyaları
Tablo 2 — TickType_t veri türü ve configTICK_TYPE_WIDTH_IN_BITS yapılandırması
Tablo 3 — Makro ön ekleri
Tablo 4 — Yaygın makro tanımları
Tablo 5 — uxBitsToWaitFor ve xWaitForAllBits parametrelerinin etkisi

FreeRTOS™ Gerçek Zamanlı Çekirdeğinde Uzmanlaşma

Mastering the FreeRTOS™ Real Time Kernel — A Hands-On Tutorial Guide

Richard Barry ve The FreeRTOS Ekibi
Çeviren: Ertan Suluağaç

1 Önsöz (Preface)

1.1 Küçük Gömülü Sistemlerde Çoklu Görev (Multitasking in Small Embedded Systems)

1.1.1 FreeRTOS Çekirdeği Hakkında

FreeRTOS, gerçek zamanlı bir çekirdek (kernel) ve tamamlayıcı işlevleri uygulayan bir dizi modüler kütüphaneden oluşan C kütüphaneleri koleksiyonudur.

Richard Barry, FreeRTOS'u ilk olarak 2003 civarında geliştirdi. Richard'ın şirketi Real-Time Engineers Ltd, Amazon Web Services (AWS) 2016'da FreeRTOS'un yönetimini devralana kadar dünyanın önde gelen çip şirketleriyle yakın ortaklık içinde FreeRTOS'u geliştirmeye devam etti. Richard şu anda FreeRTOS üzerindeki çalışmalarına AWS IoT ekibinde kıdemli baş mühendis olarak devam etmektedir. FreeRTOS, her türlü amaç için kullanılabilen, MIT lisanslı açık kaynak kodlu bir yazılımdır. AWS'nin yönetiminden faydalanmak için bir AWS müşterisi olmanıza gerek yoktur!

FreeRTOS çekirdeği, mikrodenetleyiciler veya küçük mikroişlemciler üzerinde çalışan derinlemesine gömülü (deeply embedded) gerçek zamanlı uygulamalar için son derece uygundur. Bu tür bir uygulama tipik olarak hem katı (hard) hem de esnek (soft) gerçek zamanlı gereksinimlerin bir karışımını içerir.

esnek gerçek zamanlı gereksinimler bir zaman sınırı belirtir—ancak bu sürenin aşılması sistemi işe yaramaz hale getirmez. Örneğin, tuş vuruşlarına çok yavaş yanıt vermek, sistemi gerçekten kullanılamaz hale getirmeden sinir bozucu derecede tepkisiz görünmesine neden olabilir.

katı gerçek zamanlı gereksinimler bir zaman sınırı belirtir—ve bu sürenin aşılması sistemin mutlak bir başarısızlığına neden olur. Örneğin, bir sürücü hava yastığının çarpışma sensörü girdilerine çok yavaş yanıt vermesi yarardan çok zarar verme potansiyeline sahiptir.

FreeRTOS çekirdeği, FreeRTOS üzerinde oluşturulan uygulamaların katı gerçek zamanlı gereksinimlerini karşılamasını sağlayan gerçek zamanlı bir çekirdektir (veya gerçek zamanlı çizgeleyici - scheduler). Uygulamaların bağımsız yürütme iş

parçacıkları (threads) koleksiyonu olarak düzenlenmesini sağlar. Örneğin, yalnızca bir çekirdeği olan bir işlemcide, herhangi bir anda yalnızca tek bir yürütme iş parçacığı çalışabilir. Çekirdek, uygulama tasarımcısı tarafından her bir iş parçacığına atanan önceliği (priority) inceleyerek hangi iş parçacığının yürütüleceğine karar verir. En basit durumda, uygulama tasarımcısı katı gerçek zamanlı gereksinimleri uygulayan iş parçacıklarına daha yüksek öncelikler ve esnek gerçek zamanlı gereksinimleri uygulayan iş parçacıklarına daha düşük öncelikler atayabilir. Önceliklerin bu şekilde tahsis edilmesi, katı gerçek zamanlı iş parçacıklarının her zaman esnek gerçek zamanlı iş parçacıklarından önce yürütülmesini sağlar, ancak öncelik atama kararları her zaman bu kadar basit değildir.

Önceki paragraftaki kavramları henüz tam olarak anlamadıysanız endişelenmeyin. Sonraki bölümler, gerçek zamanlı bir çekirdeği ve özellikle FreeRTOS'u nasıl kullanacağınızı anlamanıza yardımcı olacak pek çok örnekle birlikte ayrıntılı bir açıklama sunmaktadır.

1.1.2 Değer Önerisi (Value Proposition)

FreeRTOS çekirdeğinin benzeri görülmemiş küresel başarısı, ilgi çekici değer önerisinden gelmektedir; FreeRTOS profesyonel olarak geliştirilmiş, sıkı bir kalite kontrolünden geçmiş, sağlam, desteklenen, herhangi bir fikri mülkiyet sahipliği belirsizliği içermeyen ve tescilli kaynak kodunuzu ifşa etme zorunluluğu olmaksızın ticari uygulamalarda kullanımı gerçekten ücretsiz olan bir yazılımdır. Ayrıca AWS'nin yönetimi; küresel bir varlık, uzman güvenlik olayı müdahale prosedürleri, büyük ve çeşitli bir geliştirme ekibi, resmi doğrulama (formal verification), sızma testleri (pen testing), bellek güvenliği kanıtları ve uzun vadeli destek sağlarken, FreeRTOS'u donanım, geliştirme aracı ve bulut hizmetinden bağımsız bir açık kaynak projesi olarak korur. FreeRTOS geliştirme süreci GitHub'da şeffaf ve topluluk odaklıdır ve herhangi bir özel araç veya geliştirme uygulaması gerektirmez.

Bize söylemeden, hatta herhangi bir ücret ödemeden FreeRTOS kullanarak bir ürünü piyasaya sürebilirsiniz; binlerce şirket de tam olarak bunu yapmaktadır. Herhangi bir zamanda ek destek almak isterseniz veya hukuk ekibiniz ek yazılı garantiler veya tazminat (indemnification) talep ederse, stratejik ortaklarımız basit ve düşük maliyetli ticari lisans seçenekleri sunar. Seçtiğiniz herhangi bir zamanda ticari rotayı tercih edebileceğinizi bilmenin rahatlığını yaşarsınız.

1.1.3 Terminoloji Hakkında Bir Not

FreeRTOS'ta her bir yürütme iş parçacığına (thread) 'görev (task)' adı verilir. Gömülü sistemler topluluğunda terminoloji konusunda bir fikir birliği yoktur, ancak ben 'iş parçacığı (thread)' yerine 'görev (task)' terimini tercih ediyorum, çünkü iş parçacığının bazı uygulama alanlarında daha spesifik bir anlamı olabilir.

1.1.4 Neden Bir RTOS Kullanmalı?

Çoklu iş parçacıklı (multithreading) bir çekirdek kullanmadan iyi gömülü yazılımlar yazmak için çok iyi bilinen teknikler vardır. Geliştirilmekte olan sistem basitse, bu teknikler en uygun çözümü sağlayabilir. Daha karmaşık durumlarda bir çekirdek

kullanmak muhtemelen daha çok tercih edilir, ancak geçiş noktasının (crossover point) nerede oluştuğu her zaman öznel olacaktır.

Zaten tarif edildiği gibi, görev önceliklendirmesi (task prioritization) bir uygulamanın işlem sürelerini karşılamasını sağlamaya yardımcı olabilir, ancak bir çekirdek çok belirgin olmayan başka faydalar da sağlayabilir. Bunlardan bazıları aşağıda çok kısaca listelenmiştir.

- **Çizgeleme bilgisinin soyutlanması:** RTOS, yürütme çizgelemesinden sorumludur ve uygulamaya zamanla ilgili bir API sağlar. Bu, uygulama kodunun yapısının daha anlaşılır olmasını ve genel kod boyutunun daha küçük olmasını sağlar.
- **Sürdürülebilirlik/Genişletilebilirlik:** Çizgeleme ayrıntılarının soyutlanması, modüller arasındaki karşılıklı bağımlılıkların (interdependencies) azalmasıyla sonuçlanır ve yazılımın kontrollü ve öngörülebilir bir şekilde gelişmesini sağlar. Ayrıca, çekirdek çizgelemeden sorumlu olduğundan, uygulama performansı temel donanımdaki değişikliklere karşı daha az duyarlıdır.
- **Modülerlik:** Görevler bağımsız modüllerdir ve her birinin iyi tanımlanmış bir amacı olmalıdır.
- **Ekip gelişimi:** Görevlerin aynı zamanda iyi tanımlanmış arayüzleri olmalıdır, bu da ekip halinde geliştirmeyi kolaylaştırır.
- **Daha kolay test etme:** Temiz arayüzlere sahip, iyi tanımlanmış bağımsız modüller olan görevlerin izolasyon halinde test edilmesi daha kolaydır.
- **Kodun yeniden kullanımı:** Daha fazla modülerlik ve daha az karşılıklı bağımlılıkla tasarlanmış kodun yeniden kullanımı daha kolaydır.
- **Artan verimlilik:** Bir RTOS kullanan uygulama kodu tamamen olay odaklı (event-driven) olabilir. Gerçekleşmeyen olaylar için yoklama (polling) yaparak işlem süresinin boşa harcanmasına gerek yoktur. Olay odaklı olmaktan kazanılan verimliliğe karşılık, RTOS tick kesmesini (interrupt) işleme ve yürütmeyi bir görevden diğerine değiştirme ihtiyacı doğar. Ancak, bir RTOS kullanmayan uygulamalar genellikle zaten bir çeşit tick kesmesi içerir.
- **Boşta kalma süresi (Idle time):** Otomatik olarak oluşturulan Boş (Idle) görevi, işlem gerektiren hiçbir uygulama görevi olmadığında çalışır. Boş (Idle) görev yedek işlem kapasitesini ölçebilir, arka plan kontrolleri gerçekleştirebilir veya işlemciyi düşük güç moduna (low-power mode) geçirebilir.
- **Güç Yönetimi (Power Management):** Bir RTOS kullanımından kaynaklanan verimlilik kazanımları, işlemcinin düşük güç modunda daha fazla zaman geçirmesini sağlar. İşlemci, Boş (Idle) görev çalıştığı her seferinde düşük güç durumuna geçirilerek güç tüketimi önemli ölçüde azaltılabilir. FreeRTOS ayrıca özel bir 'tick-siz' (tick-less) moda sahiptir. Tick-siz modun kullanılması, işlemcinin aksi halde mümkün olandan daha düşük bir güç moduna girmesine ve düşük güç modunda daha uzun süre kalmasına olanak tanır.
- **Esnek kesme işleme (Flexible interrupt handling):** Kesme işleyicileri (Interrupt handlers), işlemlerin uygulama yazarı tarafından oluşturulan bir göreve veya otomatik olarak oluşturulan RTOS daemon görevine (zamanlayıcı görevi - timer task - olarak da bilinir) ertelenmesiyle çok kısa tutulabilir.

- **Karma işlem gereksinimleri:** Basit tasarım desenleri, bir uygulama içinde periyodik, sürekli ve olay odaklı işlemlerin bir karışımını elde edebilir. Ek olarak, uygun görev ve kesme öncelikleri seçilerek katı ve esnek gerçek zamanlı gereksinimler karşılanabilir.

1.1.5 FreeRTOS Çekirdeği Özellikleri

FreeRTOS çekirdeği aşağıdaki standart özelliklere sahiptir:

- Önleyici (Pre-emptive) veya işbirlikçi (co-operative) çalışma
- İsteğe bağlı zaman dilimleme (time-slicing)
- Çok esnek görev önceliği ataması
- Esnek, hızlı ve hafif görev bildirim mekanizmaları (task notification)
- Kuyruklar (Queues)
- İkili semaforlar (Binary semaphores)
- Sayan semaforlar (Counting semaphores)
- Muteksler (Mutexes)
- Özyinelemeli muteksler (Recursive mutexes)
- Yazılım zamanlayıcıları (Software timers)
- Olay grupları (Event groups)
- Akış arabellekleri (Stream buffers)
- Mesaj arabellekleri (Message buffers)
- Eş yordamlar (Co-routines - kullanımdan kaldırıldı)
- Tick kanca (hook) işlevleri
- Boş (Idle) kanca işlevleri
- Yığın (Stack) taşması denetimi
- İzleme (Trace) makroları
- Görev çalışma zamanı istatistikleri toplama
- İsteğe bağlı ticari lisanslama ve destek
- Tam kesme iç içe geçme (interrupt nesting) modeli (bazı mimariler için)
- Aşırı düşük güçlü uygulamalar için tick-siz yetenek (bazı mimariler için)
- Görevleri izole etmek ve uygulama güvenliğini artırmak için Bellek Koruma Birimi (Memory Protection Unit) desteği (bazı mimariler için)
- Uygun olduğunda yazılım yönetimli kesme yığını (bu, RAM'den tasarruf etmeye yardımcı olabilir)
- Statik veya dinamik olarak tahsis edilmiş bellek kullanarak RTOS nesneleri oluşturma yeteneği

1.1.6 Lisanslama ve FreeRTOS, OpenRTOS ile SafeRTOS Ailesi

FreeRTOS MIT açık kaynak lisansı şunları sağlamak üzere tasarlanmıştır:

- FreeRTOS ticari uygulamalarda kullanılabilir.

- FreeRTOS'un kendisi herkes için ücretsiz olarak erişilebilir kalır.
- FreeRTOS kullanıcıları fikri mülkiyetlerinin mülkiyetini ellerinde tutarlar.

En son açık kaynak lisans bilgileri için https://www.FreeRTOS.org/license adresine bakın.

OpenRTOS, FreeRTOS'un Amazon Web Services lisansı altında üçüncü bir tarafça sağlanan ticari lisanslı bir versiyonudur.

SafeRTOS, FreeRTOS ile aynı kullanım modelini paylaşır, ancak uluslararası alanda tanınan çeşitli güvenlikle ilgili standartlara uygunluk iddia etmek için gerekli olan pratiklere, prosedürlere ve süreçlere uygun olarak geliştirilmiştir.

1.2 Dahil Edilen Kaynak Dosyalar ve Projeler

1.2.1 Bu Kitaba Eşlik Eden Örneklerin Elde Edilmesi

https://www.FreeRTOS.org/Documentation/code adresinden indirilebilecek zip dosyası, bu kitapta sunulan örnekleri derlemek ve yürütmek için gerekli tüm kaynak kodunu, önceden yapılandırılmış proje dosyalarını ve talimatları içerir. Zip dosyasının mutlaka FreeRTOS'un en son sürümünü içermeyeceğini unutmayın.

Bu kitapta yer alan ekran görüntüleri, FreeRTOS Windows bağlantı noktasını (port) kullanarak bir Microsoft Windows ortamında yürütülen örnekleri göstermektedir. FreeRTOS Windows portunu kullanan proje, https://www.visualstudio.com/ adresinden edinilebilen Visual Studio'nun ücretsiz Community (Topluluk) sürümü kullanılarak derlenecek şekilde önceden yapılandırılmıştır. FreeRTOS Windows portunun kullanışlı bir değerlendirme, test ve geliştirme platformu sağlamasına rağmen, gerçek (true) gerçek-zamanlı davranış sağlamadığını unutmayın.

2 FreeRTOS Çekirdek Dağıtımı (The FreeRTOS Kernel Distribution)

2.1 Giriş

Kullanıcıların FreeRTOS çekirdek dosyaları ve dizinlerine aşina olmalarına yardımcı olmak için bu bölüm:

- FreeRTOS dizin yapısının üst düzey bir görünümünü sağlar.
- Belirli bir FreeRTOS projesi için gereken kaynak dosyaları açıklar.
- Demo uygulamalarını tanıtır.
- Yeni bir FreeRTOS projesinin nasıl oluşturulacağı hakkında bilgi sağlar.

Buradaki açıklama yalnızca resmi FreeRTOS dağıtımıyla ilgilidir. Bu kitapla birlikte gelen örnekler biraz farklı bir organizasyon kullanmaktadır.

2.2 FreeRTOS Dağıtımını Anlamak

2.2.1 Tanım: FreeRTOS Port (Bağlantı Noktası)

FreeRTOS yaklaşık yirmi farklı derleyici ile derlenebilir ve kırktan fazla farklı işlemci mimarisinde çalışabilir. Desteklenen her bir derleyici ve işlemci kombinasyonuna FreeRTOS portu (bağlantı noktası) adı verilir.

2.2.2 FreeRTOS'u Derlemek

FreeRTOS, aksi takdirde tek iş parçacıklı (single-threaded) ve çıplak metal (bare-metal) olacak bir uygulamaya çoklu görev (multi-tasking) yetenekleri sağlayan bir kütüphanedir.

FreeRTOS bir dizi C kaynak dosyası olarak sağlanır. Bazı kaynak dosyalar tüm portlar için ortakken, diğerleri bir porta özgüdür. Kaynak dosyaları projenizin bir parçası olarak derlemek, FreeRTOS API'sini uygulamanızın kullanımına sunar. Her resmi FreeRTOS portu için referans olarak kullanılabilecek bir demo uygulaması sağlanmıştır. Demo uygulaması, doğru kaynak dosyaları derlemek ve doğru başlık dosyalarını (header files) dahil etmek için önceden yapılandırılmıştır.

Oluşturuldukları sırada, her demo uygulaması derleyici hatası veya uyarısı olmadan 'kutudan çıktığı gibi (out of the box)' derleniyordu. Derleme araçlarında yapılan sonraki değişikliklerin artık durumun böyle olmadığı anlamına gelmesi halinde lütfen FreeRTOS destek forumlarını (https://forums.FreeRTOS.org) kullanarak bize bildirin. Bölüm 2.3 demo uygulamalarını açıklamaktadır.

2.2.3 FreeRTOSConfig.h

`FreeRTOSConfig.h` adlı bir başlık dosyasında tanımlanan sabitler çekirdeği yapılandırır. `FreeRTOSConfig.h` dosyasını doğrudan kaynak dosyalarınıza dahil etmeyin! Bunun yerine, `FreeRTOSConfig.h` dosyasını uygun zamanda dahil edecek olan `FreeRTOS.h` dosyasını dahil edin.

`FreeRTOSConfig.h`, FreeRTOS çekirdeğini belirli bir uygulamada kullanılmak üzere uyarlamak için kullanılır. Örneğin, `FreeRTOSConfig.h`, FreeRTOS'un işbirlikçi (cooperative) veya önleyici (pre-emptive) çizgeleme kullanıp kullanmadığını tanımlayan `configUSE_PREEMPTION` gibi sabitler içerir (Bölüm 4.13 çizgeleme algoritmalarını açıklamaktadır).

`FreeRTOSConfig.h`, FreeRTOS'u belirli bir uygulama için uyarlar, bu nedenle FreeRTOS kaynak kodunu içeren bir dizinde değil, uygulamanın parçası olan bir dizinde bulunmalıdır.

Ana FreeRTOS dağıtımı her FreeRTOS portu için bir demo uygulaması içerir ve her demo uygulamasının kendi `FreeRTOSConfig.h` dosyası vardır. Dosyayı sıfırdan oluşturmak yerine, kullandığınız FreeRTOS portu için sağlanan demo uygulaması tarafından kullanılan `FreeRTOSConfig.h` ile başlamanız ve ardından bunu uygulamanıza göre uyarlamanız önerilir.

FreeRTOS referans kılavuzu ve https://www.freertos.org/a00110.html, `FreeRTOSConfig.h` içinde yer alan sabitleri açıklar. Tüm sabitleri `FreeRTOSConfig.h`'ye dahil etmek gerekli değildir; birçoğu atlanırsa varsayılan bir değer alır.

2.2.4 Resmi Dağıtımlar

Çekirdek dahil olmak üzere bireysel FreeRTOS kütüphaneleri, kendi Github depolarından ve bir zip dosyası arşivi olarak edinilebilir. Bireysel kütüphaneleri elde etme yeteneği, üretim kodunda FreeRTOS kullanırken uygundur. Ancak, hem kütüphaneleri hem de örnek projeleri içerdiğinden, başlamak için ana FreeRTOS dağıtımını indirmek daha iyidir.

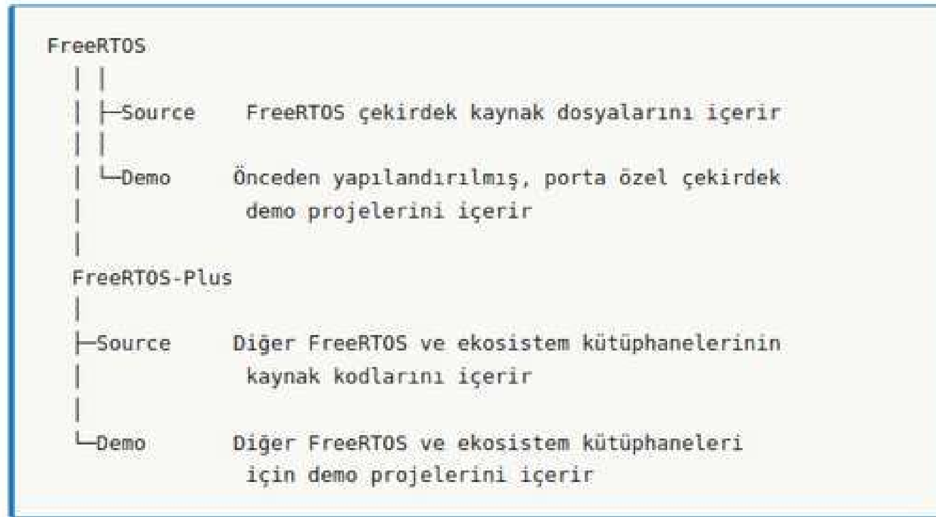
Ana dağıtım, tüm FreeRTOS kütüphaneleri için kaynak kodunu, tüm FreeRTOS çekirdek portlarını ve tüm FreeRTOS demo uygulamaları için proje dosyalarını içerir. Dosya sayısından gözünüz korkmasın! Uygulamalar yalnızca küçük bir alt kümeye ihtiyaç duyar.

En son dağıtımı içeren bir zip dosyasını indirmek için https://github.com/FreeRTOS/FreeRTOS/releases/latest adresini kullanın. Alternatif olarak, ilgili Git depolarından alt modül olarak eklenen bireysel kütüphaneler de dahil olmak üzere ana dağıtımı GitHub'dan klonlamak için aşağıdaki Git komutlarından birini kullanın:

```
git clone https://github.com/FreeRTOS/FreeRTOS.git --recurse-submodules

git clone git@github.com:FreeRTOS/FreeRTOS.git --recurse-submodules
```

Şekil 2.1, FreeRTOS dağıtımının birinci ve ikinci düzey dizinlerini göstermektedir:

Şekil 2.1 — FreeRTOS dağıtımındaki üst düzey dizinler



Dağıtım, FreeRTOS çekirdek kaynak dosyalarının yalnızca bir kopyasını içerir; tüm demo projeleri çekirdek kaynak dosyalarını `FreeRTOS/Source` dizininde bulmayı bekler ve dizin yapısı değiştirilirse derlenmeyebilir.

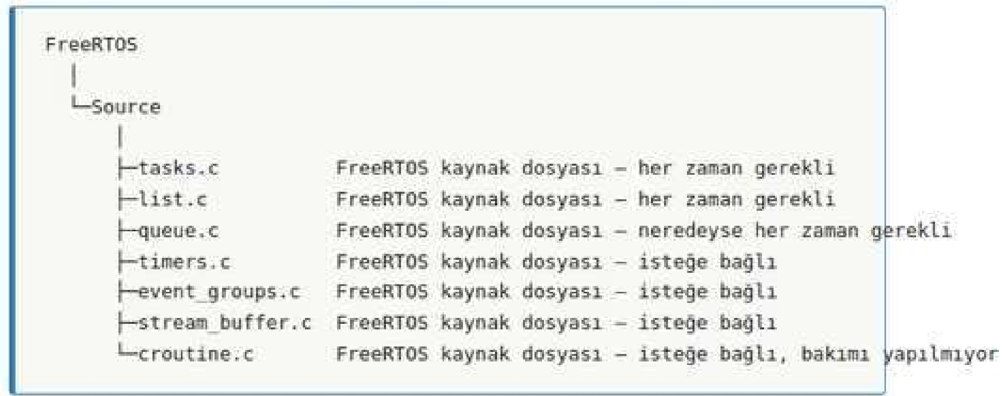
2.2.6 Tüm Portlarda Ortak Olan FreeRTOS Kaynak Dosyaları

`tasks.c` ve `list.c` çekirdek FreeRTOS kernel işlevselliğini uygular ve her zaman gereklidir. Şekil 2.2'de gösterildiği gibi doğrudan `FreeRTOS/Source` dizininde bulunurlar. Aynı dizin aşağıdaki isteğe bağlı kaynak dosyalarını da içerir:

- **`queue.c`:** `queue.c`, bu kitabın ilerleyen kısımlarında açıklandığı gibi hem kuyruk hem de semafor hizmetleri sağlar. `queue.c` neredeyse her zaman gereklidir.
- **`timers.c`:** `timers.c`, bu kitabın ilerleyen kısımlarında açıklandığı gibi yazılım zamanlayıcısı işlevselliği sağlar. Sadece uygulama yazılım zamanlayıcıları kullanıyorsa derlenmesi gerekir.
- **`event_groups.c`:** `event_groups.c`, bu kitabın ilerleyen kısımlarında açıklandığı gibi olay grubu (event group) işlevselliği sağlar. Sadece uygulama olay gruplarını kullanıyorsa derlenmesi gerekir.
- **`stream_buffer.c`:** `stream_buffer.c`, bu kitabın ilerleyen kısımlarında açıklandığı gibi hem akış arabelleği (stream buffer) hem de mesaj arabelleği (message buffer) işlevselliği sağlar. Sadece uygulama akış veya mesaj arabelleklerini kullanıyorsa derlenmesi gerekir.
- **`croutine.c`:** `croutine.c`, FreeRTOS eş-yordam (co-routine) işlevselliğini uygular. Yalnızca uygulama eş-yordamları kullanıyorsa derlenmesi gerekir. Eş-yordamların çok küçük mikrodenetleyicilerde kullanılması amaçlanmıştır, günümüzde nadiren kullanılmaktadır. Bu nedenle artık bakımları

yapılmamaktadır ve yeni tasarımlar için kullanımları önerilmemektedir. Eş-yordamlar bu kitapta anlatılmamaktadır.

Şekil 2.2 — FreeRTOS dizin ağacındaki çekirdek kaynak dosyaları



Pek çok proje zaten aynı isimlere sahip dosyaları kullanacağından, zip dosyası dağıtımında kullanılan dosya isimlerinin ad alanı çakışmalarına (namespace clashes) neden olabileceği kabul edilmektedir. Kullanıcılar gerekirse dosya isimlerini değiştirebilir, ancak isimler dağıtımda değiştirilemez, zira böyle yapmak mevcut kullanıcıların projelerinin yanı sıra FreeRTOS farkındalığı olan geliştirme araçlarıyla uyumluluğu da bozacaktır.

2.2.7 Bir Porta Özgü FreeRTOS Kaynak Dosyaları

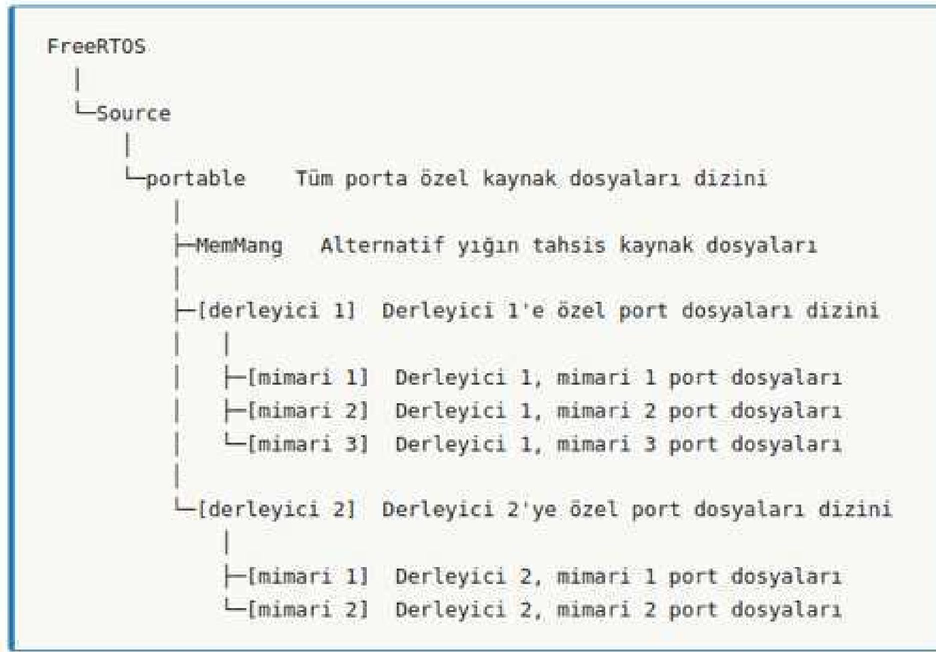
`FreeRTOS/Source/portable` dizini bir FreeRTOS portuna özgü kaynak dosyaları içerir. Taşınabilir (portable) dizini, önce derleyiciye, ardından işlemci mimarisine göre bir hiyerarşi olarak düzenlenmiştir. Şekil 2.3 hiyerarşiyi göstermektedir.

FreeRTOS'u 'compiler (derleyici)' derleyicisini kullanarak 'architecture (mimari)' mimarisine sahip bir işlemcide çalıştırmak için, çekirdek FreeRTOS kaynak dosyalarına ek olarak, `FreeRTOS/Source/portable/[compiler]/[architecture]` dizininde bulunan dosyaları da derlemelisiniz.

Bölüm 3, Yığın (Heap) Bellek Yönetimi'nde açıklandığı gibi, FreeRTOS yığın bellek tahsisini de taşınabilir (portable) katmanın bir parçası olarak kabul eder. `configSUPPORT_DYNAMIC_ALLOCATION` 0 olarak ayarlanmışsa, projenize bir yığın bellek ayırma şeması dahil etmeyin.

FreeRTOS, `FreeRTOS/Source/portable/MemMang` dizininde örnek yığın tahsis şemaları sağlar. FreeRTOS dinamik bellek tahsisi kullanacak şekilde yapılandırılmışsa, söz konusu dizindeki yığın uygulama kaynak dosyalarından birini projenize dahil etmeniz veya kendi uygulamanızı sağlamanız gerekir.

! Dikkat: Projenize örnek yığın tahsisi uygulamalarından birden fazlasını dahil etmeyin.



Şekil 2.3 — FreeRTOS dizin ağacındaki porta özel kaynak dosyalar

2.2.8 İçerme Yolları (Include Paths)

FreeRTOS, derleyicinin içerme yoluna (include path) üç dizinin dahil edilmesini gerektirir. Bunlar:

1. Çekirdek FreeRTOS kernel başlık dosyalarının (header files) yolu: `FreeRTOS/Source/include`.
2. Kullanımdaki FreeRTOS portuna özgü kaynak dosyalarının yolu: `FreeRTOS/Source/portable/[compiler]/[architecture]`.
3. Doğru `FreeRTOSConfig.h` başlık dosyasının yolu.

2.2.9 Başlık Dosyaları (Header Files)

FreeRTOS API'sini kullanan bir kaynak dosyası önce `FreeRTOS.h` dosyasını, ardından API işlevi için prototipi içeren başlık dosyasını (`task.h`, `queue.h`, `semphr.h`, `timers.h`, `event_groups.h`, `stream_buffer.h`, `message_buffer.h` veya `croutine.h`) içermelidir. Başka hiçbir FreeRTOS başlık dosyasını açıkça dahil etmeyin; `FreeRTOS.h` otomatik olarak `FreeRTOSConfig.h` dosyasını içerir.

2.3 Demo Uygulamaları

Her FreeRTOS portu, oluşturulduğu sırada, derleyici hatası veya uyarısı olmadan 'kutudan çıktığı gibi' derlenen en az bir demo uygulamasıyla birlikte gelir. Derleme araçlarında yapılan sonraki değişikliklerin artık durumun böyle olmadığı anlamına gelmesi halinde lütfen FreeRTOS destek forumlarını (https://forums.FreeRTOS.org) kullanarak bize bildirin.

Çapraz Platform Desteği: FreeRTOS, Windows, Linux ve MacOS sistemlerinde ve hem gömülü hem de geleneksel çeşitli araç zincirleriyle geliştirilir ve test edilir. Ancak ara sıra sürüm farklılıkları veya gözden kaçan bir test nedeniyle derleme hataları ortaya çıkabilir. Bu tür hataları bize bildirmek için lütfen FreeRTOS destek forumunu (https://forums.FreeRTOS.org) kullanın.

Demo uygulamalarının birkaç amacı vardır:

- Doğru dosyaların dahil edildiği ve doğru derleyici seçeneklerinin ayarlandığı, çalışan ve önceden yapılandırılmış bir proje örneği sağlamak.
- Minimum kurulum veya ön bilgi ile 'kutudan çıktığı gibi' denemelere izin vermek.
- FreeRTOS API'lerinin nasıl kullanılacağını göstermek.
- Gerçek uygulamaların oluşturulabileceği bir temel (base) olmak.
- Çekirdeğin uygulamasını stres testine tabi tutmak.

Her demo projesi `FreeRTOS/Demo` dizini altında benzersiz bir alt dizinde bulunur. Alt dizinin adı, demo projesinin ilgili olduğu portu gösterir.

FreeRTOS.org web sitesi her bir demo uygulaması için bir sayfa içerir. Web sayfası aşağıdaki konularda bilgi içerir:

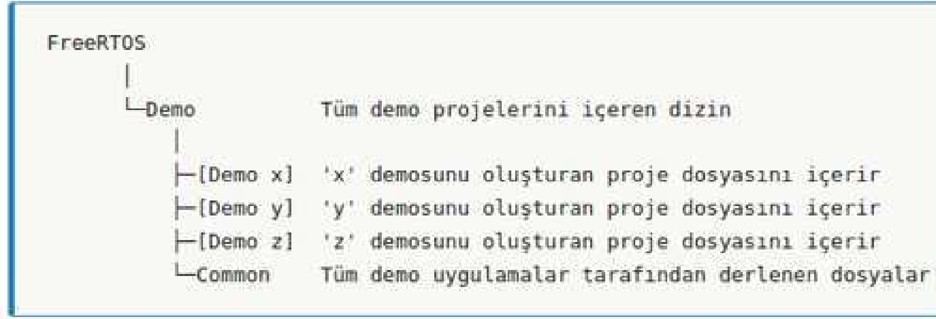
- FreeRTOS dizin yapısı içinde demo için proje dosyasının nasıl bulunacağı.
- Projenin kullanmak üzere yapılandırıldığı donanım veya emülatör.
- Demoyu çalıştırmak için donanımın nasıl kurulacağı.
- Demonun nasıl derleneceği.
- Demonun beklenen davranışı.

Tüm demo projeleri, uygulamaları `FreeRTOS/Demo/Common/Minimal` dizininde bulunan 'ortak demo görevlerinin (common demo tasks)' bir alt kümesini oluşturur. Ortak demo görevleri, FreeRTOS API'sinin nasıl kullanılacağını göstermek ve FreeRTOS çekirdek portlarını test etmek için vardır; belirli bir yararlı işlevsellik uygulamazlar.

Pek çok demo projesi, tipik olarak iki RTOS görevi ve bir kuyruk oluşturan basit bir 'blinky' tarzı başlangıç projesi oluşturacak şekilde de yapılandırılabilir.

Her demo projesi, FreeRTOS çekirdeğini başlatmadan önce demo uygulaması görevlerini oluşturan `main()` işlevini içeren `main.c` adlı bir dosya içerir. O demoya özel bilgiler için her bir `main.c` dosyası içindeki yorumlara (comments) bakın.

Şekil 2.4 — Demo dizin hiyerarşisi



2.4 FreeRTOS Projesi Oluşturma

2.4.1 Sağlanan Demo Projelerinden Birini Uyarlama

Her FreeRTOS portu önceden yapılandırılmış en az bir demo uygulamasıyla birlikte gelir. Yeni projenin doğru dosyaları içerdiğinden, doğru kesme işleyicilerinin (interrupt handlers) yüklendiğinden ve doğru derleyici seçeneklerinin ayarlandığından emin olmak için yeni projelerin bu mevcut projelerden biri uyarlanarak oluşturulması önerilir.

Mevcut bir demo projesinden yeni bir uygulama oluşturmak için:

1. Sağlanan demo projesini açın ve beklendiği gibi derlendiğinden ve çalıştığından emin olun.
2. `Demo/Common` dizininde bulunan dosyalar olan, demo görevlerini uygulayan kaynak dosyaları kaldırın.
3. Liste 2.1'de gösterildiği gibi, `prvSetupHardware()` ve `vTaskStartScheduler()` hariç, `main()` içindeki tüm işlev çağrılarını (function calls) silin.
4. Projenin hala derlendiğini doğrulayın.

Bu adımları izlediğinizde doğru FreeRTOS kaynak dosyalarını içeren, ancak herhangi bir işlevsellik tanımlamayan bir proje oluşturmuş olursunuz.

```
int main( void )
{
    /* Gerekli donanım kurulumlarını yapın. */
    prvSetupHardware();

    /* --- UYGULAMA GÖREVLERİ BURADA OLUŞTURULABİLİR --- */

    /* Oluşturulan görevleri çalıştırmaya başlayın. */
    vTaskStartScheduler();

    /* Yürütme yalnızca çizgeleyiciyi başlatmak için yeterli
       yığın (heap) olmadığında buraya ulaşacaktır. */
    for( ;; );
    return 0;
}
```

Liste 2.1 Yeni bir `main()` işlevi için şablon

2.4.2 Sıfırdan Yeni Bir Proje Oluşturma

Daha önce de belirtildiği gibi, yeni projelerin mevcut bir demo projesinden oluşturulması tavsiye edilir. Eğer bu arzu edilmiyorsa, yeni bir proje oluşturmak için aşağıdaki prosedürü kullanın:

1. Seçtiğiniz araç zincirini (toolchain) kullanarak, henüz hiçbir FreeRTOS kaynak dosyasını içermeyen yeni bir proje oluşturun.
2. Yeni projenin derlendiğinden, hedef donanımınıza yüklendiğinden (downloads) ve çalıştığından emin olun.
3. Ancak zaten çalışan bir projeniz olduğundan emin olduğunuzda, Tablo 1'de ayrıntıları verilen FreeRTOS kaynak dosyalarını projeye ekleyin.
4. Demo proje tarafından kullanılan ve kullanımdaki port için sağlanan `FreeRTOSConfig.h` başlık dosyasını yeni proje dizininize kopyalayın.
5. Projenin başlık dosyalarını bulmak için arayacağı yola (path) aşağıdaki dizinleri ekleyin:
    - `FreeRTOS/Source/include`

    - `FreeRTOS/Source/portable/[compiler]/[architecture]` (burada [compiler] ve [architecture] seçtiğiniz port için doğru olanlardır)
    - `FreeRTOSConfig.h` başlık dosyasını içeren dizin
6. Derleyici ayarlarını ilgili demo projesinden kopyalayın.
7. Gerekli olabilecek herhangi bir FreeRTOS kesme işleyicisini (interrupt handler) kurun. Referans olarak, kullanımdaki portu tanımlayan web sayfasını ve kullanımdaki port için sağlanan demo projesini kullanın.

Tablo 1 Projeye dahil edilecek FreeRTOS kaynak dosyaları

Dosya (File)	Konum (Location)
`tasks.c`	`FreeRTOS/Source`
`queue.c`	`FreeRTOS/Source`
`list.c`	`FreeRTOS/Source`
`timers.c`	`FreeRTOS/Source`
`event_groups.c`	`FreeRTOS/Source`
`stream_buffer.c`	`FreeRTOS/Source`
Tüm C ve assembler dosyaları	`FreeRTOS/Source/portable/[compiler]/[architecture]`
`heap_n.c`	`FreeRTOS/Source/portable/MemMang` (burada n, 1, 2, 3, 4 veya 5'tir)

Yığın (heap) belleği üzerine not: `configSUPPORT_DYNAMIC_ALLOCATION` 0 ise, projenize bir yığın bellek tahsis şeması dahil etmeyin. Aksi takdirde projenize, `heap_n.c` dosyalarından birini veya kendiniz tarafından sağlanan bir yığın bellek ayırma şemasını dahil edin. Daha fazla bilgi için Bölüm 3, Yığın Bellek Yönetimi'ne başvurun.

2.5 Veri Tipleri ve Kodlama Stili Kılavuzu

2.5.1 Veri Tipleri

FreeRTOS'un her portu, (diğer şeylerin yanı sıra) iki porta özgü veri tipi için tanımlar içeren benzersiz bir `portmacro.h` başlık dosyasına sahiptir: `TickType_t` ve `BaseType_t`. Aşağıdaki liste, kullanılan makro veya typedef'i ve gerçek türü açıklamaktadır:

`TickType_t`

- FreeRTOS, tick kesmesi (tick interrupt) adı verilen periyodik bir kesme yapılandırır.
- FreeRTOS uygulaması başladığından beri meydana gelen tick kesmelerinin sayısına tick count (tick sayısı) denir. Tick sayısı zamanın bir ölçüsü olarak kullanılır.
- İki tick kesmesi arasındaki süreye tick period (tick periyodu) denir. Zamanlar tick periyotlarının katları olarak belirtilir.
- `TickType_t`, tick sayısı değerini tutmak ve zamanları belirtmek için kullanılan veri türüdür.
- `TickType_t`, `FreeRTOSConfig.h` içindeki `configTICK_TYPE_WIDTH_IN_BITS` ayarına bağlı olarak işaretsiz 16 bitlik bir tür, işaretsiz 32 bitlik bir tür veya işaretsiz 64 bitlik bir tür olabilir. `configTICK_TYPE_WIDTH_IN_BITS` ayarı mimariye bağlıdır. FreeRTOS portları ayrıca bu ayarın geçerli olup olmadığını kontrol edecektir.
- 16 bitlik bir tür kullanmak, 8 bitlik ve 16 bitlik mimarilerde verimliliği büyük ölçüde artırabilir, ancak FreeRTOS API çağrılarında belirtilebilecek maksimum blok (block) süresini ciddi şekilde sınırlar. 32-bit veya 64-bit mimaride 16-bit `TickType_t` türü kullanmak için hiçbir neden yoktur.
- Önceki `configUSE_16_BIT_TICKS` kullanımı, 32-bit'ten daha büyük tick sayılarını desteklemek için `configTICK_TYPE_WIDTH_IN_BITS` ile değiştirilmiştir. Yeni tasarımlarda `configUSE_16_BIT_TICKS` yerine `configTICK_TYPE_WIDTH_IN_BITS` kullanılmalıdır.

Tablo 2 `TickType_t` veri tipi ve `configTICK_TYPE_WIDTH_IN_BITS` konfigürasyonu

configTICK_TYPE_WIDTH_IN_BITS	8-bit mimariler	16-bit mimariler	32-bit mimariler	64-bit mimariler
TICK_TYPE_WIDTH_16_BITS	uint16_t	uint16_t	uint16_t	N/A
TICK_TYPE_WIDTH_32_BITS	uint32_t	uint32_t	uint32_t	N/A

TICK_TYPE_WIDTH_64_BITS	N/A	N/A	uint64_t	uint64_t

`BaseType_t`

- Bu her zaman mimari için en verimli veri tipi olarak tanımlanır. Tipik olarak bu, 64 bitlik bir mimaride 64 bitlik bir tür, 32 bitlik bir mimaride 32 bitlik bir tür, 16 bitlik bir mimaride 16 bitlik bir tür ve 8 bitlik bir mimaride 8 bitlik bir türdür.
- `BaseType_t` genel olarak yalnızca çok sınırlı bir değer aralığı alan dönüş tipleri için ve `pdTRUE`/`pdFALSE` tipli Boolean'lar için kullanılır.

2.5.2 Değişken İsimleri

Değişkenlerin başına türleri eklenir: char için 'c', `int16_t` (short) için 's', `int32_t` (long) için 'l' ve `BaseType_t` ile diğer standart dışı türler (yapılar (structures), görev tanıtıcıları (task handles), kuyruk tanıtıcıları (queue handles) vb.) için 'x'.

Bir değişken işaretsiz (unsigned) ise, başına bir 'u' da eklenir. Bir değişken işaretçi (pointer) ise, başına bir 'p' de eklenir. Örneğin, `uint8_t` türünde bir değişken 'uc' ile başlarken, char işaretçisi (`char *`) türünde bir değişken 'pc' ile başlar.

2.5.3 İşlev (Fonksiyon) İsimleri

İşlevlerin başına hem döndürdükleri tür hem de içinde tanımlandıkları dosya eklenir. Örneğin:

- `vTaskPrioritySet()` bir `void` döndürür ve `tasks.c` içinde tanımlanmıştır.
- `xQueueReceive()` `BaseType_t` türünde bir değişken döndürür ve `queue.c` içinde tanımlanmıştır.
- `pvTimerGetTimerID()` `void` türünde bir işaretçi (pointer to `void`) döndürür ve `timers.c` içinde tanımlanmıştır.

Dosya kapsamındaki (özel - private) işlevler 'prv' önekiyle başlar.

2.5.4 Biçimlendirme

Sekmeler (Tabs), bir sekmenin her zaman dört boşluğa (space) eşit olduğu bazı demo uygulamalarında kullanılır. Çekirdek artık sekmeleri kullanmamaktadır.

2.5.5 Makro İsimleri

Çoğu makro büyük harflerle yazılır ve başlarına makronun nerede tanımlandığını gösteren küçük harfler eklenir. Tablo 3 öneklerin bir listesini sunmaktadır.

Tablo 3 Makro önekleri

Önek (Prefix)	Makro tanımının konumu
port (örneğin, `portMAX_DELAY`)	`portable.h` veya `portmacro.h`
task (örneğin, `taskENTER_CRITICAL()`)	`task.h`
pd (örneğin, `pdTRUE`)	`projdefs.h`
config (örneğin, `configUSE_PREEMPTION`)	`FreeRTOSConfig.h`
err (örneğin, `errQUEUE_FULL`)	`projdefs.h`

Semafor API'sinin neredeyse tamamen bir dizi makro olarak yazıldığını, ancak makro isimlendirme kuralından ziyade işlev (fonksiyon) isimlendirme kuralını izlediğini unutmayın.

Tablo 4'te tanımlanan makrolar FreeRTOS kaynak kodu boyunca kullanılır.

Tablo 4 Ortak makro tanımları

Makro (Macro)	Değer (Value)
`pdTRUE`	1
`pdFALSE`	0
`pdPASS`	1
`pdFAIL`	0

2.5.6 Aşırı Tür Dönüştürme (Type Casting) Mantığı

FreeRTOS kaynak kodu, birçoğu uyarıları nasıl ve ne zaman ürettikleri konusunda farklılık gösteren birçok farklı derleyici ile derlenir. Özellikle, farklı derleyiciler tür dönüştürmenin (casting) farklı şekillerde kullanılmasını ister. Sonuç olarak, FreeRTOS kaynak kodu normalde gerekli olandan daha fazla tür dönüştürme içerir.

3 Yığın (Heap) Bellek Yönetimi (Heap Memory Management)

3.1 Giriş

3.1.1 Ön Koşullar

Yetkin bir C programcısı olmak FreeRTOS kullanmanın bir ön koşuludur, bu nedenle bu bölüm okuyucunun aşağıdaki gibi kavramlara aşina olduğunu varsayar:

- Bir C projesi oluşturmanın farklı derleme ve bağlama (linking) aşamaları.
- Yığın (stack) ve heap'in (yığın bellek) ne olduğu.
- Standart C kütüphanesi `malloc()` ve `free()` işlevleri.

3.1.2 Kapsam

Bu bölüm şunları kapsar:

- FreeRTOS'un RAM'i ne zaman tahsis ettiği.
- FreeRTOS ile sağlanan beş örnek bellek tahsis şeması.
- Hangi bellek tahsis şemasının seçileceği.

3.1.3 Statik ve Dinamik Bellek Tahsisi Arasında Geçiş Yapmak

Aşağıdaki bölümlerde görevler, kuyruklar, semaforlar ve olay grupları gibi çekirdek nesneleri tanıtılmaktadır. Bu nesneleri tutmak için gereken RAM, derleme zamanında statik olarak veya çalışma zamanında dinamik olarak tahsis edilebilir. Dinamik tahsis, tasarım ve planlama çabasını azaltır, API'yi basitleştirir ve RAM ayak izini (footprint) en aza indirir. Statik tahsis daha deterministiktir (öngörülebilirdir), bellek tahsis hatalarını ele alma ihtiyacını ortadan kaldırır ve yığın parçalanması riskini (heap fragmentation - yığının yeterli boş belleğe sahip olduğu ancak bunun kullanılabilir tek bir bitişik blokta olmadığı durum) ortadan kaldırır.

Statik olarak tahsis edilmiş bellek kullanarak çekirdek nesneleri oluşturan FreeRTOS API işlevleri, yalnızca `FreeRTOSConfig.h` dosyasında `configSUPPORT_STATIC_ALLOCATION` değeri 1 olarak ayarlandığında kullanılabilir. Dinamik olarak tahsis edilmiş bellek kullanarak çekirdek nesneleri oluşturan FreeRTOS API işlevleri, yalnızca `FreeRTOSConfig.h` dosyasında `configSUPPORT_DYNAMIC_ALLOCATION` değeri 1 olarak ayarlandığında veya tanımsız bırakıldığında kullanılabilir. Her iki sabitin aynı anda 1 olarak ayarlanması geçerlidir.

`configSUPPORT_STATIC_ALLOCATION` ile ilgili daha fazla bilgi bölüm 3.4 Statik Bellek Tahsisi Kullanımı içinde bulunmaktadır.

3.1.4 Dinamik Bellek Tahsisini Kullanmak

Dinamik bellek tahsisi bir C programlama kavramıdır, FreeRTOS'a veya çoklu göreve özgü bir kavram değildir. Çekirdek nesnelerinin isteğe bağlı olarak dinamik olarak tahsis edilen bellek kullanılarak oluşturulabilmesi ve genel amaçlı C kütüphanesi `malloc()` ve `free()` işlevlerinin aşağıdaki nedenlerden biri veya birkaçı nedeniyle uygun olmaması sebebiyle FreeRTOS ile ilgilidir:

- Küçük gömülü sistemlerde her zaman mevcut değildirler.
- Uygulamaları nispeten büyük olabilir ve değerli kod alanını kaplayabilir.
- Nadiren iş parçacığı güvenlidirler (thread-safe).
- Deterministik değildirler; işlevleri yürütmek için geçen süre çağrıdan çağrıya farklılık gösterecektir.
- Parçalanmadan (fragmentation) muzdarip olabilirler (yığının yeterli boş belleğe sahip olduğu ancak kullanılabilir tek bir bitişik blokta olmadığı durum).
- Bağlayıcı (linker) yapılandırmasını karmaşıklaştırabilirler.
- Yığın alanının diğer değişkenler tarafından kullanılan belleğe doğru büyümesine izin verilirse, hata ayıklaması zor olan hataların kaynağı olabilirler.

3.1.5 Dinamik Bellek Tahsisi için Seçenekler

FreeRTOS'un ilk sürümleri, farklı boyutlardaki bellek blokları havuzlarının derleme zamanında önceden tahsis edildiği ve ardından bellek ayırma işlevleri tarafından geri döndürüldüğü bir bellek havuzları (memory pools) ayırma şeması kullanıyordu. Blok ayırma gerçek zamanlı sistemlerde yaygın olmasına rağmen, gerçekten küçük gömülü sistemlerde RAM'in verimsiz kullanımı birçok destek talebine yol açtığından FreeRTOS'tan çıkarılmıştır.

FreeRTOS artık bellek tahsisini (çekirdek kod tabanının bir parçası yerine) taşınabilir katmanın (portable layer) bir parçası olarak ele almaktadır. Bunun nedeni, farklı gömülü sistemlerin farklı dinamik bellek tahsisi ve çizgeleme gereksinimlerine sahip olmasıdır, bu nedenle tek bir dinamik bellek tahsisi algoritması her zaman uygulamaların yalnızca bir alt kümesi için uygun olacaktır. Ayrıca, dinamik bellek tahsisinin çekirdek kod tabanından çıkarılması, uygulama yazarlarının uygun olduğunda kendi özel uygulamalarını sağlamalarına olanak tanır.

FreeRTOS RAM'e ihtiyaç duyduğunda `malloc()` yerine `pvPortMalloc()` işlevini çağırır. Benzer şekilde, FreeRTOS daha önce ayrılmış olan RAM'i serbest bıraktığında `free()` yerine `vPortFree()` işlevini çağırır. `pvPortMalloc()` standart C kütüphanesi `malloc()` işleviyle aynı prototipe ve `vPortFree()` standart C kütüphanesi `free()` işleviyle aynı prototipe sahiptir.

`pvPortMalloc()` ve `vPortFree()` genel (public) işlevlerdir, bu nedenle uygulama kodundan da çağrılabilirler.

FreeRTOS, tümü bu bölümde belgelenmiş olan `pvPortMalloc()` ve `vPortFree()` için beş örnek uygulama ile birlikte gelir. FreeRTOS uygulamaları bu örnek uygulamalardan birini kullanabilir veya kendi uygulamalarını sağlayabilir.

Beş örnek sırasıyla `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` ve `heap_5.c` kaynak dosyalarında tanımlanmıştır ve tümü `FreeRTOS/Source/portable/MemMang` dizininde bulunur.

3.2 Örnek Bellek Tahsis Şemaları

3.2.1 Heap_1

Küçük, amaca özel gömülü sistemlerin FreeRTOS zamanlayıcısını başlatmadan önce yalnızca görevler ve diğer çekirdek nesneleri oluşturması yaygındır. Durum böyle olduğunda, bellek yalnızca uygulama herhangi bir gerçek zamanlı işlevsellik gerçekleştirmeye başlamadan önce çekirdek tarafından (dinamik olarak) tahsis edilir ve bellek uygulamanın ömrü boyunca tahsis edilmiş olarak kalır. Bu, seçilen tahsis şemasının determinizm ve parçalanma gibi daha karmaşık bellek tahsis sorunlarını dikkate alması gerekmediği ve bunun yerine kod boyutu ve basitlik gibi niteliklere öncelik verebileceği anlamına gelir.

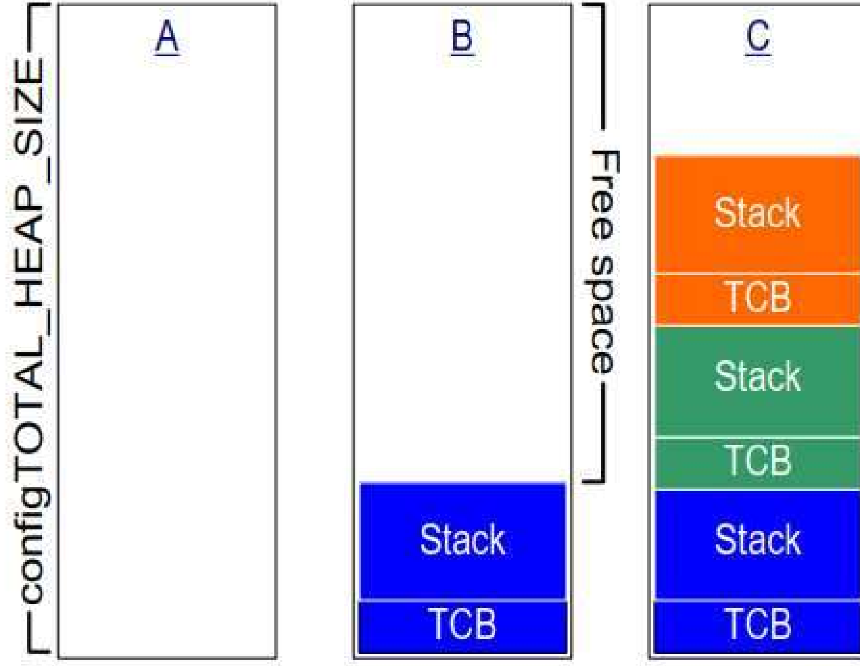
`Heap_1.c`, `pvPortMalloc()` işlevinin çok temel bir sürümünü uygular ve `vPortFree()` işlevini uygulamaz. Hiçbir zaman bir görevi veya diğer çekirdek nesnelerini silmeyen uygulamalar `heap_1` kullanma potansiyeline sahiptir. Dinamik bellek tahsisi kullanımını aksi takdirde yasaklayacak olan bazı ticari açıdan kritik ve güvenlik açısından kritik sistemler de `heap_1` kullanma potansiyeline sahiptir. Kritik sistemler, deterministik olmama durumu, bellek parçalanması ve başarısız tahsislerle ilişkili belirsizlikler nedeniyle dinamik bellek tahsisini sıklıkla yasaklar. `Heap_1` her zaman deterministiktir ve belleği parçalayamaz.

`Heap_1`'in `pvPortMalloc()` uygulaması, çağrıldığı her seferinde FreeRTOS yığını (heap) adı verilen basit bir `uint8_t` dizisini (array) daha küçük bloklara böler. `FreeRTOSConfig.h` içindeki `configTOTAL_HEAP_SIZE` sabiti dizinin boyutunu bayt cinsinden ayarlar. Yığının statik olarak ayrılmış bir dizi olarak uygulanması, FreeRTOS'un çok fazla RAM tüketiyormuş gibi görünmesine neden olur çünkü yığın FreeRTOS verisinin bir parçası haline gelir.

Dinamik olarak tahsis edilen her görev `pvPortMalloc()`'a iki çağrı ile sonuçlanır. İlki bir görev kontrol bloğu (Task Control Block - `TCB`) ve ikincisi de görevin yığınını (stack) tahsis eder. Şekil 3.1, görevler oluşturuldukça `heap_1`'in basit diziyi nasıl alt bölümlere ayırdığını göstermektedir.

Şekil 3.1'e bakıldığında:

- A, hiçbir görev oluşturulmadan önceki diziyi gösterir—tüm dizi boştur.
- B, bir görev oluşturulduktan sonraki diziyi gösterir.
- C, üç görev oluşturulduktan sonraki diziyi gösterir.



Şekil 3.1 Her görev oluşturulduğunda heap_1 dizisinden RAM tahsis edilmesi

3.2.2 Heap_2

`Heap_2`'nin yerini gelişmiş işlevsellik içeren `heap_4` almıştır. `Heap_2`, geriye dönük uyumluluk (backward compatibility) için FreeRTOS dağıtımında tutulmaktadır ve yeni tasarımlar için önerilmemektedir.

`Heap_2.c` de `configTOTAL_HEAP_SIZE` sabiti ile boyutlandırılan bir diziyi bölerek çalışır. Bellek tahsis etmek için "en uygun (best-fit)" algoritmasını kullanır ve `heap_1`'in aksine `vPortFree()` işlevini uygular. Yine, yığının (heap) statik olarak tahsis edilmiş bir dizi olarak uygulanması, yığın FreeRTOS verisinin bir parçası haline geldiği için FreeRTOS'un çok fazla RAM tüketiyormuş gibi görünmesine neden olur.

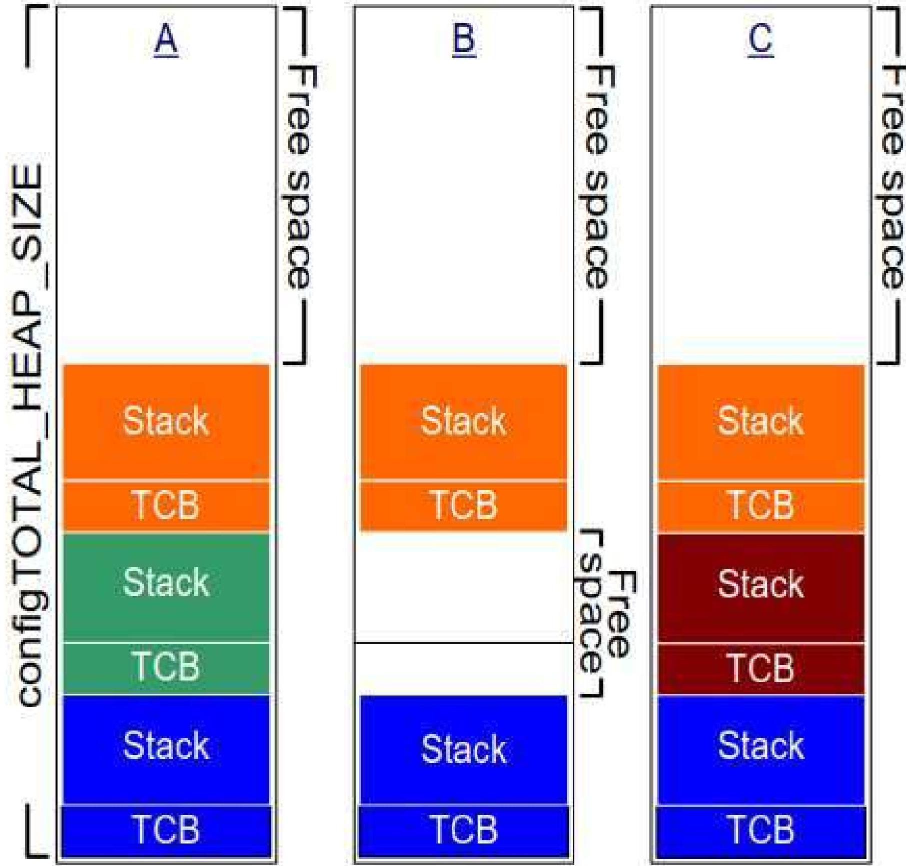
En uygun algoritması (best-fit), `pvPortMalloc()`'un istenen bayt sayısına boyut olarak en yakın olan boş bellek bloğunu kullanmasını sağlar. Örneğin, şu senaryoyu ele alalım:

- Yığın, sırasıyla 5 bayt, 25 bayt ve 100 baytlık üç serbest bellek bloğu içeriyor.
- `pvPortMalloc()`, 20 bayt RAM talep ediyor.

İstenen bayt sayısının sığdığı en küçük serbest RAM bloğu 25 baytlık bloktur, bu nedenle `pvPortMalloc()` 20 baytlık bloğa bir işaretçi döndürmeden önce 25 baytlık bloğu 20 baytlık bir blok ve 5 baytlık bir blok olarak böler. Yeni 5 baytlık blok, gelecekteki `pvPortMalloc()` çağrıları için kullanılabilir kalır.

(Not: Bu aşırı basitleştirilmiş bir örnektir, çünkü heap_2 yığın alanındaki blok boyutları hakkında bilgi saklar, bu nedenle bölünen iki bloğun toplamı aslında 25'ten az olacaktır.)

heap_4'ün aksine, heap_2 bitişik serbest blokları tek bir büyük blok halinde birleştirmez (coalesce yapmaz), bu nedenle parçalanmaya (fragmentation) heap_4'ten daha duyarlıdır. Ancak, tahsis edilen ve ardından serbest bırakılan bloklar her zaman aynı boyutta ise parçalanma bir sorun değildir.



Şekil 3.2 Görevler oluşturuldukça ve silindikçe heap_2 dizisinden RAM'in ayrılması ve serbest bırakılması

Şekil 3.2, bir görev oluşturulduğunda, silindiğinde ve yeniden oluşturulduğunda en uygun algoritmasının nasıl çalıştığını göstermektedir. Şekil 3.2'ye bakıldığında:

- A, üç görev tahsis edildikten sonra diziyi gösterir. Dizinin üst kısmında büyük bir boş blok kalır.
- B, görevlerden biri silindikten sonraki diziyi gösterir. Dizinin üst kısmındaki büyük boş blok kalır. Ayrıca daha önce silinen görevin TCB'sini ve yığınını (stack) tutan daha küçük iki boş blok da vardır.
- C, başka bir görev oluşturulduktan sonraki durumu gösterir. Görevin oluşturulması, xTaskCreate() API işlevi içinden pvPortMalloc()'a biri yeni bir TCB, diğeri de görev yığınını ayırmak için olmak üzere iki çağrı yapılmasıyla sonuçlanmıştır.

Her TCB aynı boyuttadır, bu nedenle en uygun algoritması oluşturulan görevin TCB'sini tutmak için silinen görevin TCB'sini tutan RAM bloğunu yeniden kullanır.

Yeni oluşturulan göreve tahsis edilen yığının (stack) boyutu, daha önce silinen göreve tahsis edilenle aynı boyuttaysa, en uygun algoritması, oluşturulan görevin yığınını tutmak için silinen görevin yığınını tutan RAM bloğunu yeniden kullanır.

Dizinin en üstündeki daha büyük tahsis edilmemiş blok dokunulmamış olarak kalır.

`Heap_2` deterministik değildir, ancak `malloc()` ve `free()`'nin çoğu standart kütüphane uygulamasından daha hızlıdır.

3.2.3 Heap_3

`Heap_3.c` standart kütüphane `malloc()` ve `free()` işlevlerini kullanır, bu nedenle bağlayıcı (linker) yapılandırması yığın (heap) boyutunu tanımlar ve `configTOTAL_HEAP_SIZE` sabiti kullanılmaz.

`Heap_3`, yürütülmeleri süresince FreeRTOS zamanlayıcısını (scheduler) geçici olarak askıya alarak `malloc()` ve `free()`'yi iş parçacığı güvenli (thread-safe) hale getirir.

3.2.4 Heap_4

`heap_1` ve `heap_2` gibi, `heap_4` de bir diziyi daha küçük bloklara ayırarak çalışır. Daha önce olduğu gibi, dizi statik olarak ayrılır ve `configTOTAL_HEAP_SIZE` ile boyutlandırılır, bu da yığın FreeRTOS verisinin bir parçası haline geldiği için FreeRTOS'un çok fazla RAM kullandığı görünümünü verir.

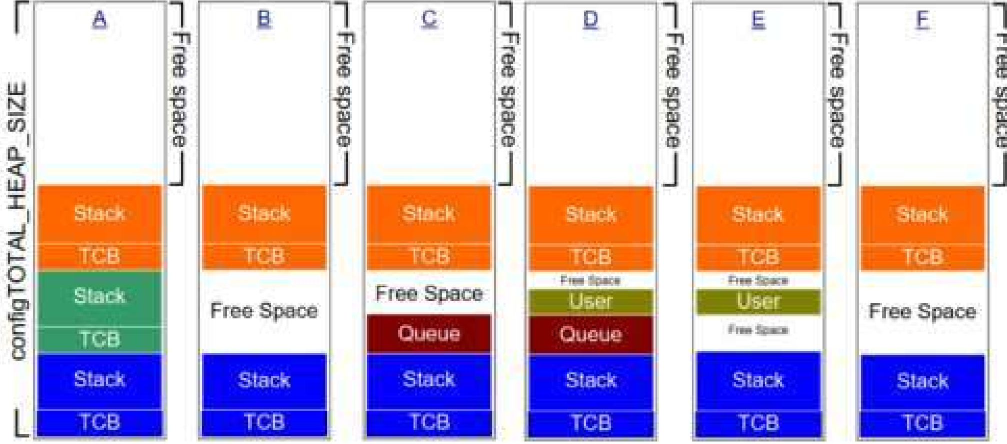
`Heap_4` bellek tahsis etmek için "ilk uygun (first-fit)" algoritmasını kullanır. `Heap_2`'nin aksine, `heap_4` bitişik (adjacent) serbest bellek bloklarını tek bir daha büyük blok halinde birleştirir (coalesces), bu da bellek parçalanması riskini en aza indirir.

İlk uygun (first-fit) algoritması, `pvPortMalloc()`'un talep edilen bayt sayısını tutacak kadar büyük olan ilk serbest bellek bloğunu kullanmasını sağlar. Örneğin, şu senaryoyu ele alalım:

- Yığın, dizide görünme sırasına göre sırasıyla 5 bayt, 200 bayt ve 100 bayt olan üç serbest bellek bloğu içeriyor.
- `pvPortMalloc()` 20 bayt RAM talep ediyor.

İstenen bayt sayısının sığdığı ilk boş RAM bloğu 200 baytlık bloktur, bu nedenle `pvPortMalloc()` 20 baytlık bloğa bir işaretçi döndürmeden önce 200 baytlık bloğu 20 baytlık ve 180 baytlık bir blok olmak üzere ikiye böler. Yeni 180 baytlık blok, gelecekteki `pvPortMalloc()` çağrıları için mevcut kalır.

`Heap_4`, bitişik boş blokları birleştirerek (coalescing) parçalanma riskini en aza indirir ve farklı boyutlardaki RAM bloklarını tekrar tekrar ayıran ve serbest bırakan uygulamalar için uygun hale getirir.



Şekil 3.3 heap_4 dizisinden tahsis edilen ve serbest bırakılan RAM

Şekil 3.3 bellek birleştirme ile heap_4 ilk-uygun algoritmasının nasıl çalıştığını göstermektedir. Şekil 3.3'e bakıldığında:

- A, üç görev oluşturduktan sonraki diziyi gösterir. Dizinin üst kısmında büyük bir boş blok kalır.
- B, görevlerden biri silindikten sonraki diziyi gösterir. Dizinin üst kısmındaki büyük boş blok kalır. Artık silinen görevin TCB'sinin ve yığınının (stack) eskiden bulunduğu yerde başka bir boş blok var. heap_2 örneğinden farklı olarak, heap_4, silinen görevin sırasıyla TCB ve yığınını tutan iki bellek bloğunu birleştirerek daha büyük tek bir boş blok haline getirir.
- C, bir FreeRTOS kuyruğu oluşturulduktan sonraki durumu gösterir. Kuyrukları dinamik olarak ayırmak için kullanılan `xQueueCreate()` API işlevi, kuyruğun kullandığı RAM'i ayırmak için `pvPortMalloc()`'u çağırır. heap_4 ilk uygun algoritmasını kullandığından, `pvPortMalloc()`, Şekil 3.3'te görevin silinmesiyle boşaltılan RAM olan, kuyruğu tutacak kadar büyük olan ilk boş RAM bloğundan RAM tahsis eder. Kuyruk, boş bloktaki tüm RAM'i tüketmez, bu nedenle blok ikiye bölünür ve kullanılmayan kısım gelecekteki `pvPortMalloc()` çağrıları için uygun kalır.
- D, bir FreeRTOS API işlevini çağırmak yerine doğrudan uygulama kodundan `pvPortMalloc()`'u çağırdıktan sonraki durumu gösterir. Kullanıcının tahsis ettiği blok, kuyruğa tahsis edilen bellek ile ondan sonraki TCB'ye tahsis edilen bellek arasındaki blok olan ilk boş bloğa sığacak kadar küçüktü. Görevin silinmesiyle serbest kalan bellek şimdi üç ayrı bloğa bölünmüştür; ilk blok kuyruğu tutar, ikinci blok kullanıcının tahsis ettiği belleği tutar ve üçüncü blok boş kalır.
- E, silinen kuyruğa ayrılan belleği otomatik olarak serbest bırakan kuyruğu sildikten sonraki durumu gösterir. Artık kullanıcının tahsis ettiği bloğun her iki tarafında da boş bellek var.
- F, kullanıcının tahsis ettiği belleği serbest bıraktıktan sonraki durumu gösterir. Kullanıcının tahsis ettiği blok tarafından önceden kullanılan bellek, daha büyük bir tek serbest blok oluşturmak için her iki taraftaki boş bellekle birleştirilmiştir.

`Heap_4` deterministik değildir, ancak `malloc()` ve `free()`'nin çoğu standart kütüphane uygulamasından daha hızlıdır.

3.2.5 Heap_5

`Heap_5`, `heap_4` ile aynı tahsis algoritmasını kullanır. Yalnızca tek bir diziden bellek ayırmakla sınırlı olan `heap_4`'ün aksine, `heap_5` birbirinden ayrı birden çok bellek alanındaki belleği tek bir yığında (heap) birleştirebilir. `Heap_5`, FreeRTOS'un çalıştığı sistem tarafından sağlanan RAM sistemin bellek haritasında tek bir bitişik (boşluksuz) blok olarak görünmediğinde kullanışlıdır.

3.2.6 heap_5'i Başlatma (Initialising): vPortDefineHeapRegions() API İşlevi

`vPortDefineHeapRegions()`, `heap_5` tarafından yönetilen yığını oluşturan her ayrı bellek alanının başlangıç adresini ve boyutunu belirterek `heap_5`'i başlatır. `Heap_5`, açık başlatma gerektiren tek yığın tahsisi şemasıdır ve `vPortDefineHeapRegions()` çağrılana kadar kullanılamaz. Bu, `vPortDefineHeapRegions()` çağrılana kadar görevler, kuyruklar ve semaforlar gibi çekirdek nesnelerinin dinamik olarak oluşturulamayacağı anlamına gelir.

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions );
```

Liste 3.1 `vPortDefineHeapRegions()` API işlev prototipi

`vPortDefineHeapRegions()` tek parametresi olarak bir `HeapRegion_t` yapıları dizisi (array of structures) alır. Her yapı, yığının parçası olacak bir bellek bloğunun başlangıç adresini ve boyutunu tanımlar — yapıların tüm dizisi yığın alanının tamamını tanımlar.

```
typedef struct HeapRegion

{

    /* Yığının (heap) bir parçası olacak bellek bloğunun başlangıç adresi. */

    uint8_t *pucStartAddress;

    /* Bellek bloğunun bayt cinsinden boyutu. */

    size_t xSizeInBytes;

} HeapRegion_t;
```

Liste 3.2 `HeapRegion_t` yapısı

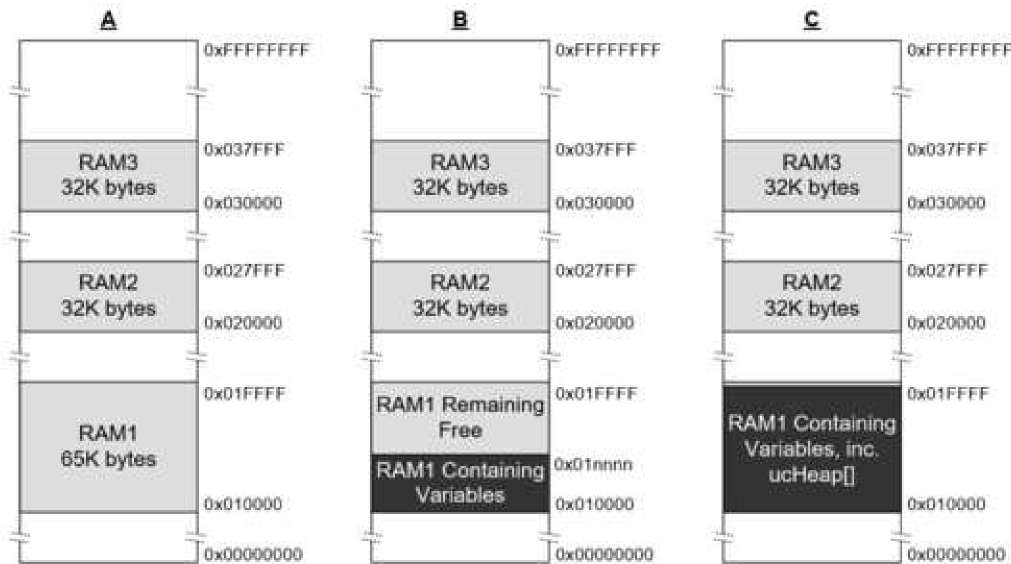
Parametreler:

pxHeapRegions: `HeapRegion_t` yapıları dizisinin başlangıcına giden bir işaretçi. Her bir yapı, yığının bir parçası olacak bellek bloğunun başlangıç adresini ve boyutunu tanımlar.

Dizideki `HeapRegion_t` yapıları başlangıç adresine göre sıralanmalıdır; en düşük başlangıç adresine sahip bellek alanını tanımlayan `HeapRegion_t` yapısı dizideki ilk yapı olmalıdır ve en yüksek başlangıç adresine sahip bellek alanını tanımlayan `HeapRegion_t` yapısı dizideki son yapı olmalıdır.

Dizinin sonunu, `pucStartAddress` üyesi `NULL` olarak ayarlanmış bir `HeapRegion_t` yapısıyla işaretleyin.

Örnek olarak, üç ayrı RAM bloğu (RAM1, RAM2 ve RAM3) içeren Şekil 3.4 A'da gösterilen varsayımsal bellek haritasını ele alalım. Yürütülebilir kodun, gösterilmeyen salt okunur belleğe (read-only memory) yerleştirildiği varsayılmaktadır.



Şekil 3.4 Bellek Haritası

Liste 3.3, birlikte üç RAM bloğunu bütünüyle tanımlayan bir `HeapRegion_t` yapıları dizisini göstermektedir.

```
/* Üç RAM bölgesinin başlangıç adresini ve boyutunu tanımlayın. */

#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x00010000 )

#define RAM1_SIZE ( 64 * 1024 )

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )
```

```
#define RAM3_SIZE ( 32 * 1024 )


/* Üç RAM bölgesinin her biri için bir dizin içeren bir HeapRegion_t tanımları

dizisi oluşturun ve diziyi NULL adresi içeren bir HeapRegion_t yapısıyla
sonlandırın.

HeapRegion_t yapıları başlangıç adresi sırasına göre görünmelidir ve

en düşük başlangıç adresini içeren yapı ilk sırada yer almalıdır. */

const HeapRegion_t xHeapRegions[] =

{

    { RAM1_START_ADDRESS, RAM1_SIZE },

    { RAM2_START_ADDRESS, RAM2_SIZE },

    { RAM3_START_ADDRESS, RAM3_SIZE },

    { NULL, 0 } /* Dizinin sonunu işaretler. */

};


int main( void )

{

    /* heap_5'i başlat. */

    vPortDefineHeapRegions( xHeapRegions );


    /* Uygulama kodunu buraya ekleyin. */

}
```

Liste 3.3 3 RAM bölgesini bütünüyle birlikte tanımlayan `HeapRegion_t` yapıları dizisi

Liste 3.3 RAM'i doğru bir şekilde tanımlasa da, kullanılabilir bir örnek göstermez çünkü tüm RAM'i yığına (heap) tahsis eder ve diğer değişkenlerin kullanımı için hiç boş RAM bırakmaz.

Oluşturma işleminin bağlama (linking) aşaması, her değişkene bir RAM adresi ayırır. Bağlayıcı (linker) tarafından kullanılabilecek RAM normalde bağlayıcı komut dosyası (linker script) gibi bir bağlayıcı yapılandırma dosyası ile tanımlanır. Şekil 3.4 B'de bağlayıcı betiğinin RAM1 hakkında bilgi içerdiği, ancak RAM2 veya RAM3 hakkında bilgi içermediği varsayılmaktadır. Sonuç olarak bağlayıcı değişkenleri RAM1'e

yerleştirerek RAM1'in yalnızca 0x0001nnnn adresinin üzerindeki bölümünü `heap_5`'in kullanımına bırakmıştır. 0x0001nnnn adresinin gerçek değeri, uygulamada yer alan tüm değişkenlerin birleşik boyutuna bağlıdır. Bağlayıcı, tüm RAM2'yi ve tüm RAM3'ü kullanılmamış halde bırakarak RAM2'nin tamamını ve RAM3'ün tamamını `heap_5`'in kullanımı için kullanılabilir bırakmıştır.

Liste 3.3'te gösterilen kod, `heap_5`'e tahsis edilen 0x0001nnnn adresinin altındaki RAM'in değişkenleri tutmak için kullanılan RAM ile örtüşmesine (overlap) neden olacaktır. `xHeapRegions[]` dizisindeki ilk `HeapRegion_t` yapısının başlangıç adresini 0x00010000 yerine 0x0001nnnn olarak ayarlarsanız, yığın bağlayıcı tarafından kullanılan RAM ile örtüşmeyecektir. Ancak bu, aşağıdaki nedenlerden dolayı önerilen bir çözüm değildir:

- Başlangıç adresinin belirlenmesi kolay olmayabilir.
- Bağlayıcı tarafından kullanılan RAM miktarı gelecekteki derlemelerde değişebilir ve bu da `HeapRegion_t` yapısında kullanılan başlangıç adresinin güncellenmesini gerektirir.
- Derleme araçları, bağlayıcı tarafından kullanılan RAM ile `heap_5` tarafından kullanılan RAM'in örtüşüp örtüşmediğini bilemez ve bu nedenle uygulama yazarını uyaramaz.

Liste 3.4 daha kullanışlı ve bakımı yapılabilir bir örneği göstermektedir. `ucHeap` adında bir dizi bildirir. `ucHeap` normal bir değişkendir, bu nedenle bağlayıcı tarafından RAM1'e ayrılan verilerin bir parçası haline gelir. `xHeapRegions` dizisindeki ilk `HeapRegion_t` yapısı, `ucHeap` dizisinin başlangıç adresini ve boyutunu açıklar, böylece `ucHeap`, `heap_5` tarafından yönetilen belleğin bir parçası haline gelir. Şekil 3.4 C'de gösterildiği gibi, bağlayıcı tarafından kullanılan RAM RAM1'in tamamını tüketene kadar `ucHeap` boyutu artırılabilir.

```
/* Bağlayıcı tarafından kullanılmayan iki RAM bölgesinin başlangıç adresini ve
boyutunu tanımlayın. */

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )



/* heap_5 tarafından kullanılan yığının bir parçası olacak bir dizi bildirin. Dizi
bağlayıcı tarafından RAM1'e yerleştirilecektir. */

#define RAM1_HEAP_SIZE ( 30 * 1024 )

static uint8_t ucHeap[ RAM1_HEAP_SIZE ];



/* HeapRegion_t tanımlarından oluşan bir dizi oluşturun. Liste 3.5'te ilk giriş
tüm RAM1'i tanımlamıştır,
```

```
bu nedenle heap_5 tüm RAM1'i kullanmış olacaktır; bu kez ilk giriş yalnızca ucHeap
dizisini tanımlamaktadır,

bu nedenle heap_5 yalnızca RAM1'in ucHeap dizisini içeren kısmını kullanacaktır.

HeapRegion_t yapıları yine de en düşük başlangıç adresini içeren yapı ilk sırada
olacak şekilde

başlangıç adresi sırasına göre görünmelidir. */

const HeapRegion_t xHeapRegions[] =

{

    { ucHeap, RAM1_HEAP_SIZE },

    { RAM2_START_ADDRESS, RAM2_SIZE },

    { RAM3_START_ADDRESS, RAM3_SIZE },

    { NULL, 0 } /* Dizinin sonunu işaretler. */

};
```

Liste 3.4 Tüm RAM2'yi, tüm RAM3'ü, ancak RAM1'in yalnızca bir kısmını tanımlayan `HeapRegion_t` yapıları dizisi

Liste 3.4'te gösterilen tekniğin avantajları şunlardır:

- Sabit kodlanmış (hard-coded) bir başlangıç adresi kullanmak gerekli değildir.
- `HeapRegion_t` yapısında kullanılan adres bağlayıcı tarafından otomatik olarak ayarlanacaktır, bu nedenle bağlayıcı tarafından kullanılan RAM miktarı gelecekteki yapılarda değişse bile her zaman doğru olacaktır.
- `Heap_5`'e tahsis edilen RAM'in, bağlayıcı tarafından RAM1'e yerleştirilen verilerle çakışması imkansızdır.
- `ucHeap` çok büyükse uygulama bağlanmayacaktır (link hatası verecektir).

3.3 Yığın (Heap) ile İlgili Yardımcı İşlevler ve Makrolar

3.3.1 Yığın Başlangıç Adresini Tanımlama

`Heap_1`, `heap_2` ve `heap_4`, `configTOTAL_HEAP_SIZE` ile boyutlandırılmış statik olarak ayrılmış bir diziden bellek tahsis eder. Bu bölüm, bu tahsis şemalarını topluca `heap_n` olarak adlandırmaktadır.

Bazen yığının belirli bir bellek adresine yerleştirilmesi gerekir. Örneğin, dinamik olarak oluşturulan bir göreve atanan yığın (stack) heap'ten (yığın bellek) gelir, bu nedenle heap'i yavaş harici bellek yerine hızlı dahili belleğe yerleştirmek gerekebilir. (Görev yığınlarını hızlı bellekte tahsis etmenin başka bir yöntemi için aşağıdaki 'Görev

Yığınlarını Hızlı Belleğe Yerleştirme' alt bölümüne bakın). `configAPPLICATION_ALLOCATED_HEAP` derleme zamanı yapılandırma sabiti, uygulamanın aksi takdirde `heap_n.c` kaynak dosyasında yer alacak olan bildirimin yerine diziyi kendi kodunda bildirmesini sağlar. Diziyi uygulama kodunda bildirmek, uygulama yazarının dizinin başlangıç adresini belirtmesini sağlar.

Eğer `FreeRTOSConfig.h` dosyasında `configAPPLICATION_ALLOCATED_HEAP` 1 olarak ayarlanmışsa veya tanımsız bırakılmışsa, FreeRTOS kullanan uygulamanın `ucHeap` adlı ve `configTOTAL_HEAP_SIZE` sabitiyle boyutlandırılmış bir `uint8_t` dizisi ayırması gerekir.

Bir değişkeni belirli bir bellek adresine yerleştirmek için gereken sözdizimi, kullanımda olan derleyiciye bağlıdır, bu nedenle derleyicinizin belgelerine başvurun. İki derleyici için örnekler aşağıdadır:

- Liste 3.5 diziyi bildirmek ve diziyi `.my_heap` adlı bir bellek bölümüne yerleştirmek için GCC derleyicisinin gerektirdiği sözdizimini (syntax) göstermektedir.
- Liste 3.6, diziyi bildirmek ve diziyi `0x20000000` mutlak (absolute) bellek adresine yerleştirmek için IAR derleyicisinin gerektirdiği sözdizimini göstermektedir.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( ( section( ".my_heap" ) )
);
```

Liste 3.5 `heap_4` tarafından kullanılacak diziyi bildirmek ve diziyi `.my_heap` adlı bir bellek bölümüne yerleştirmek için GCC sözdizimi kullanımı

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

Liste 3.6 `heap_4` tarafından kullanılacak diziyi bildirmek ve diziyi `0x20000000` mutlak adresine yerleştirmek için IAR sözdizimi kullanımı

3.3.2 xPortGetFreeHeapSize() API İşlevi

`xPortGetFreeHeapSize()` API işlevi, işlev çağrıldığı sırada yığındaki serbest bayt sayısını döndürür. Yığın parçalanması hakkında bilgi vermez.

`xPortGetFreeHeapSize()`, `heap_3` için uygulanmamıştır.

```
size_t xPortGetFreeHeapSize( void );
```

Liste 3.7 `xPortGetFreeHeapSize()` API işlev prototipi

Dönüş değeri: `xPortGetFreeHeapSize()` çağrıldığı sırada yığında (heap) tahsis edilmeden kalan bayt sayısını döndürür.

3.3.3 xPortGetMinimumEverFreeHeapSize() API İşlevi

`xPortGetMinimumEverFreeHeapSize()` API işlevi, FreeRTOS uygulaması yürütülmeye başladığından beri yığında var olan minimum tahsis edilmemiş bayt sayısını döndürür.

`xPortGetMinimumEverFreeHeapSize()` tarafından döndürülen değer, uygulamanın yığın alanının tükenmesine ne kadar yaklaştığını gösterir. Örneğin, `xPortGetMinimumEverFreeHeapSize()` 200 değerini döndürürse, uygulama yürütülmeye başladığından beri bir noktada yığın alanının tükenmesine 200 bayt yaklaşmıştır.

`xPortGetMinimumEverFreeHeapSize()` yığın boyutunu optimize etmek için de kullanılabilir. Örneğin, en yüksek yığın kullanımına sahip olduğunu bildiğiniz kodu yürüttükten sonra `xPortGetMinimumEverFreeHeapSize()` 2000 değerini döndürürse, `configTOTAL_HEAP_SIZE` 2000 bayta kadar azaltılabilir.

`xPortGetMinimumEverFreeHeapSize()` yalnızca `heap_4` ve `heap_5`'te uygulanır.

```
size_t xPortGetMinimumEverFreeHeapSize( void );
```

Liste 3.8 `xPortGetMinimumEverFreeHeapSize()` API işlev prototipi

Dönüş değeri: `xPortGetMinimumEverFreeHeapSize()` FreeRTOS uygulaması çalışmaya başladığından beri yığında var olan minimum tahsis edilmemiş bayt sayısını döndürür.

3.3.4 vPortGetHeapStats() API İşlevi

`Heap_4` ve `heap_5`, işlevin tek parametresi olarak referans geçişi (pass by reference) ile `HeapStats_t` yapısını dolduran `vPortGetHeapStats()` işlevini uygular.

Liste 3.9 `vPortGetHeapStats()` işlevinin prototipini göstermektedir. Liste 3.10 ise `HeapStats_t` yapısının üyelerini göstermektedir.

```
void vPortGetHeapStats( HeapStats_t *xHeapStats );
```

Liste 3.9 `vPortGetHeapStatus()` API işlev prototipi

```
/* vPortGetHeapStats() işlevinin prototipi. */

void vPortGetHeapStats( HeapStats_t *xHeapStats );



/* HeapStats_t yapısının tanımı. Tüm boyutlar bayt cinsinden belirtilmiştir. */

typedef struct xHeapStats

{
```

```
    /* Şu anda kullanılabilir olan toplam yığın boyutu - bu, en büyük
kullanılabilir bloğun değil,

       tüm serbest blokların toplamıdır. */

    size_t xAvailableHeapSpaceInBytes;



    /* vPortGetHeapStats() çağrıldığı sırada yığın içindeki en büyük boş bloğun
boyutu. */

    size_t xSizeOfLargestFreeBlockInBytes;



    /* vPortGetHeapStats() çağrıldığı sırada yığın içindeki en küçük boş bloğun
boyutu. */

    size_t xSizeOfSmallestFreeBlockInBytes;



    /* vPortGetHeapStats() çağrıldığı sırada yığın içindeki boş bellek bloklarının
sayısı. */

    size_t xNumberOfFreeBlocks;



    /* Sistem açıldığından beri yığında kalan minimum toplam boş bellek miktarı

       (tüm boş blokların toplamı). */

    size_t xMinimumEverFreeBytesRemaining;



    /* Geçerli bir bellek bloğu döndüren pvPortMalloc() çağrılarının sayısı. */

    size_t xNumberOfSuccessfulAllocations;



    /* Bir bellek bloğunu başarıyla serbest bırakan vPortFree() çağrılarının
sayısı. */

    size_t xNumberOfSuccessfulFrees;

} HeapStats_t;
```

Liste 3.10 `HeapStatus_t()` yapısı

3.3.5 Görev Başına Yığın Kullanım İstatistiklerini Toplama

Bu kitabın TBD-RB bölümünde belgelenen `vTaskGetInfo()` API işlevi, bir `TaskStatus_t` yapısını bir görev hakkındaki bilgilerle doldurur. `FreeRTOSConfig.h` dosyasında `configTRACK_TASK_MEMORY_ALLOCATIONS` derleme zamanı sabiti 1 olarak ayarlanırsa yapı aşağıdaki ek bilgileri içerir:

- Görevin `pvPortMalloc()` işlevini çağırma sayısı.
- Görevin `vPortFree()` işlevini çağırma sayısı.
- `vTaskGetInfo()` çağrıldığı sırada herhangi bir görev tarafından henüz serbest bırakılmamış, görev tarafından tahsis edilen yığın baytlarının sayısı.
- Görev başladığından bu yana herhangi bir zamanda görev tarafından tahsis edilen maksimum yığın belleği miktarı.

3.3.6 Malloc Başarısızlık Kanca (Hook) İşlevleri

Standart kütüphane `malloc()` işlevi gibi, talep edilen RAM miktarını ayıramazsa `pvPortMalloc()` da `NULL` döndürür. Malloc başarısızlık kancası (veya geri çağırması - callback), `pvPortMalloc()` `NULL` döndürürse çağrılan uygulama tarafından sağlanan bir işlevdir. Geri çağırmanın (callback) gerçekleşmesi için `FreeRTOSConfig.h` dosyasında `configUSE_MALLOC_FAILED_HOOK` öğesini 1 olarak ayarlamalısınız. Malloc başarısızlık kancası, bir çekirdek nesnesi oluşturmak için dinamik bellek tahsisi kullanan bir FreeRTOS API işlevi içinde çağrılırsa nesne oluşturulmaz.

Eğer `FreeRTOSConfig.h` dosyasında `configUSE_MALLOC_FAILED_HOOK` 1 olarak ayarlanmışsa, uygulamanın Liste 3.11'de gösterilen isme ve prototipe sahip bir malloc başarısızlık kanca işlevi sağlaması gerekir. Uygulama bu işlevi, uygulama için uygun olan herhangi bir şekilde uygulayabilir. Sağlanan FreeRTOS demo uygulamalarının birçoğu tahsisat başarısızlığını (allocation failure) ölümcül bir hata (fatal error) olarak ele alır, ancak bu durum üretim sistemleri için en iyi uygulama değildir; tahsis hatalarından sonra zarif bir şekilde kurtulmalıdırlar (gracefully recover).

```
void vApplicationMallocFailedHook( void );
```

Liste 3.11 Malloc başarısız kancası işlev adı ve prototipi

3.3.7 Görev Yığınlarını (Task Stacks) Hızlı Belleğe Yerleştirme

Yığınlara (stacks) yüksek oranda yazıldığı ve okunduğu için, hızlı belleğe yerleştirilmeleri gerekir, ancak yığının (heap) bulunmasını istediğiniz yer burası olmayabilir. FreeRTOS, FreeRTOS API kodu içinde tahsis edilen yığınların kendi bellek ayırıcılarına (memory allocator) sahip olmalarını isteğe bağlı olarak etkinleştirmek için `pvPortMallocStack()` ve `vPortFreeStack()` makrolarını kullanır. Yığının (stack), `pvPortMalloc()` tarafından yönetilen yığından (heap) gelmesini istiyorsanız, sırasıyla `pvPortMalloc()` ve `vPortFree()` işlevlerini çağırmaya varsayılan olduklarından `pvPortMallocStack()` ve `vPortFreeStack()` makrolarını tanımsız bırakın. Aksi takdirde, Liste 3.12'de gösterildiği gibi uygulama tarafından sağlanan işlevleri çağıracak makroları tanımlayın.

```
/* Uygulama yazarı tarafından sağlanan ve hızlı bir RAM alanından bellek tahsis
eden ve

   serbest bırakan işlevler. */

void *pvMallocFastMemory( size_t xWantedSize );

void vPortFreeFastMemory( void *pvBlockToFree );



/* pvPortMallocStack() ve vPortFreeStack() makrolarını hızlı bellek kullanan
işlevlerle

   eşlemek için FreeRTOSConfig.h dosyasına aşağıdakini ekleyin. */

#define pvPortMallocStack( x ) pvMallocFastMemory( x )

#define vPortFreeStack( x ) vPortFreeFastMemory( x )
```

Liste 3.12 `pvPortMallocStack()` ve `vPortFreeStack()` makrolarının uygulama tarafından tanımlanan bir bellek ayırıcıya eşlenmesi

3.4 Statik Bellek Tahsisi Kullanımı

Bölüm 3.1.4, dinamik bellek tahsisiyle birlikte gelen bazı dezavantajları listelemektedir. Bu sorunlardan kaçınmak için statik bellek tahsisi, geliştiricinin uygulamanın ihtiyaç duyduğu her bellek bloğunu açıkça (explicity) oluşturmasına olanak tanır. Bunun aşağıdaki avantajları vardır:

- Gerekli tüm bellek derleme zamanında (compile time) bilinir.
- Tüm bellek deterministiktir.

Başka avantajları da vardır, ancak bu avantajlarla birlikte birkaç komplikasyon gelir. Ana komplikasyon, bazı çekirdek belleğini yönetmek için birkaç ek kullanıcı işlevinin eklenmesi ve ikinci komplikasyon, tüm statik belleğin uygun bir kapsamda (scope) bildirilmesini sağlama gerekliliğidir.

3.4.1 Statik Bellek Tahsisini Etkinleştirme

Statik bellek tahsisi, `FreeRTOSConfig.h` dosyasında `configSUPPORT_STATIC_ALLOCATION` öğesi 1 olarak ayarlanarak etkinleştirilir. Bu yapılandırma etkinleştirildiğinde, çekirdek işlevlerin tüm statik sürümlerini etkinleştirir. Bunlar şunlardır:

- `xTaskCreateStatic`
- `xEventGroupCreateStatic`
- `xEventGroupGetStaticBuffer`
- `xQueueGenericCreateStatic`
- `xQueueGenericGetStaticBuffers`

- `xQueueCreateMutexStatic` (if `configUSE_MUTEXES` 1 ise)
- `xQueueCreateCountingSemaphoreStatic` (if `configUSE_COUNTING_SEMAPHORES` 1 ise)
- `xStreamBufferGenericCreateStatic`
- `xStreamBufferGetStaticBuffers`
- `xTimerCreateStatic` (if `configUSE_TIMERS` 1 ise)
- `xTimerGetStaticBuffer` (if `configUSE_TIMERS` 1 ise)

Bu işlevler bu kitaptaki uygun bölümlerde açıklanacaktır.

3.4.2 Statik Dahili Çekirdek Belleği

Statik bellek ayırıcı etkinleştirildiğinde, boş (idle) görev ve zamanlayıcı (timer) görevi (etkinleştirilmişse) kullanıcı işlevleri tarafından sağlanan statik belleği kullanacaktır. Bu kullanıcı işlevleri şunlardır:

- `vApplicationGetTimerTaskMemory` (if `configUSE_TIMERS` 1 ise)
- `vApplicationGetIdleTaskMemory`

3.4.2.1 vApplicationGetTimerTaskMemory

Eğer hem `configSUPPORT_STATIC_ALLOCATION` hem de `configUSE_TIMERS` etkinleştirilmişse, çekirdek, uygulamanın zamanlayıcı görev `TCB`'si ve zamanlayıcı görev yığını (stack) için bir bellek arabelleği (memory buffer) oluşturmasına ve döndürmesine izin vermek için `vApplicationGetTimerTaskMemory()` işlevini çağıracaktır. İşlev ayrıca zamanlayıcı görev yığınının boyutunu da döndürecektir. Zamanlayıcı görev belleği işlevi için önerilen bir uygulama Liste 3.13'te gösterilmektedir.

```
void vApplicationGetTimerTaskMemory( StaticTask_t **ppxTimerTaskTCBBuffer,

                                     StackType_t **ppxTimerTaskStackBuffer,

                                     uint32_t *pulTimerTaskStackSize )

{

    /* Eğer Boş (Idle) göreve sağlanacak arabellekler (buffers) bu işlevin içinde
bildirilirse,

       statik olarak bildirilmelidirler - aksi takdirde yığın (stack) üzerinde
tahsis edilirler

       ve bu nedenle bu işlevden çıkıldıktan sonra var olmazlar. */

    static StaticTask_t xTimerTaskTCB;

    static StackType_t uxTimerTaskStack[ configMINIMAL_STACK_SIZE ];
```

```
    /* Boş görevin durumunun saklanacağı StaticTask_t yapısına bir işaretçi verin.
*/

    *ppxTimerTaskTCBBuffer = &xTimerTaskTCB;


    /* Boş görevin yığını (stack) olarak kullanılacak diziyi verin. */

    *ppxTimerTaskStackBuffer = uxTimerTaskStack;


    /* *ppxIdleTaskStackBuffer tarafından işaret edilen dizinin yığın boyutunu
verin.

       Not: yığın boyutu StackType_t sayımıdır */

    *pulTimerTaskStackSize = sizeof(uxTimerTaskStack) / sizeof(*uxTimerTaskStack);

}
```

Liste 3.13 `vApplicationGetTimerTaskMemory`'nin tipik uygulaması

SMP dahil olmak üzere herhangi bir sistemde yalnızca tek bir zamanlayıcı (timer) görevi olduğundan, zamanlayıcı görev belleği sorununa geçerli bir çözüm, `vApplicationGetTimeTaskMemory()` işlevinde statik arabellekler ayırmak ve arabellek işaretçilerini çekirdeğe geri döndürmektir.

3.4.2.2 vApplicationGetIdleTaskMemory

Boş (idle) görev, bir çekirdeğin programlanmış işi bittiğinde çalıştırılır. Boş görev bazı temizlik işleri (housekeeping) gerçekleştirir ve eğer etkinleştirilmişse kullanıcının `vTaskIdleHook()` işlevini de tetikleyebilir. Simetrik çok işlemcili bir sistemde (SMP), kalan çekirdeklerin her biri için temizlik dışı boş görevler de vardır, ancak bunlar dahili olarak `configMINIMUM_STACK_SIZE` baytlarına statik olarak tahsis edilir.

`vApplicationGetIdleTaskMemory` işlevi, uygulamanın "ana" boş görev için gereken arabellekleri oluşturmasına izin vermek için çağrılır. Liste 3.14, gerekli arabellekleri oluşturmak için statik yerel değişkenler kullanan `vApplicationIdleTaskMemory()` işlevinin tipik bir uygulamasını göstermektedir.

```
void vApplicationGetIdleTaskMemory( StaticTask_t **ppxIdleTaskTCBBuffer,

                                    StackType_t **ppxIdleTaskStackBuffer,

                                    uint32_t *pulIdleTaskStackSize )

{

    static StaticTask_t xIdleTaskTCB;

    static StackType_t uxIdleTaskStack[ configMINIMAL_STACK_SIZE ];
```

```
    *ppxIdleTaskTCBBuffer = &xIdleTaskTCB;

    *ppxIdleTaskStackBuffer = uxIdleTaskStack;

    *pulIdleTaskStackSize = configMINIMAL_STACK_SIZE;

}
```

Liste 3.14 `vApplicationGetIdleTaskMemory`'nin tipik uygulaması

4 Görev Yönetimi (Task Management)

4.1 Giriş

4.1.1 Kapsam

Bu bölüm şunları kapsamaktadır:

- FreeRTOS'un bir uygulamadaki her bir göreve işlem süresini nasıl tahsis ettiği.
- FreeRTOS'un herhangi bir zamanda hangi görevin yürütüleceğini nasıl seçtiği.
- Her bir görevin göreceli önceliğinin sistem davranışını nasıl etkilediği.
- Bir görevin bulunabileceği durumlar (states).

Bu bölümde ayrıca şunlar tartışılmaktadır:

- Görevlerin nasıl uygulanacağı (implemente edileceği).
- Bir görevin bir veya daha fazla örneğinin (instance) nasıl oluşturulacağı.
- Görev parametresinin nasıl kullanılacağı.
- Daha önce oluşturulmuş bir görevin önceliğinin nasıl değiştirileceği.
- Bir görevin nasıl silineceği.
- Bir görev kullanılarak periyodik işlemenin nasıl uygulanacağı. (İleriki bir bölüm aynı işlemin yazılım zamanlayıcıları - software timers - kullanılarak nasıl yapılacağını açıklamaktadır.)
- Boş (idle) görevin ne zaman yürütüleceği ve nasıl kullanılabileceği.

Bu bölümde sunulan kavramlar, FreeRTOS'un nasıl kullanılacağını ve FreeRTOS uygulamalarının nasıl davrandığını anlamak için temel teşkil eder. Bu nedenle, bu kitapta yer alan en detaylı bölüm budur.

4.2 Görev İşlevleri (Task Functions)

Görevler C işlevleri (fonksiyonları) olarak uygulanır. Görevler, Liste 4.1'de gösterilen, bir `void` işaretçi parametresi alan ve `void` döndüren beklenen işlev prototipini uygulamalıdır.

```
void vATaskFunction( void * pvParameters );
```

Liste 4.1 Görev işlevi prototipi

Her görev başlı başına küçük bir programdır. Bir giriş noktası vardır, normalde sonsuz bir döngü içinde sonsuza kadar çalışır ve çıkış yapmaz. Liste 4.2 tipik bir görevin yapısını göstermektedir.

Bir FreeRTOS görevinin, kendisini uygulayan işlevden herhangi bir şekilde dönmesine (return) izin verilmemelidir. Bir 'return' ifadesi içermemeli ve kendisini uygulayan işlevin sonunu geçerek yürütülmesine izin verilmemelidir. Bir göreve artık ihtiyaç duyulmuyorsa, Liste 4.2'de gösterildiği gibi açıkça (explicitly) silinmelidir.

Tek bir görev işlevi tanımı, oluşturulan her görevin ayrı bir yürütme örneği (execution instance) olduğu herhangi bir sayıda görev oluşturmak için kullanılabilir. Her örneğin kendi yığını (stack) ve dolayısıyla görevin kendi içinde tanımlanan tüm otomatik (yığın) değişkenlerinin kendi kopyası vardır.

```
void vATaskFunction( void * pvParameters )

{

    /*

     * Yığında tahsis edilen (stack-allocated) değişkenler normalde bir

     * işlevin içindeyken bildirilebilir.

     * Bu örnek işlev kullanılarak oluşturulan bir görevin her bir örneği,

     * görevin kendi yığınında tahsis edilmiş ayrı bir lStackVariable örneğine
sahip olacaktır.

     */

    long lStackVariable = 0;



    /*

     * Yığında tahsis edilen değişkenlerin aksine, `static` anahtar kelimesi ile

     * bildirilen değişkenler bağlayıcı (linker) tarafından bellekte belirli bir

     * konuma tahsis edilir.

     * Bu, vATaskFunction'ı çağıran tüm görevlerin aynı lStaticVariable örneğini

     * paylaşacağı anlamına gelir.

     */

    static long lStaticVariable = 0;



    for( ;; )

    {
```

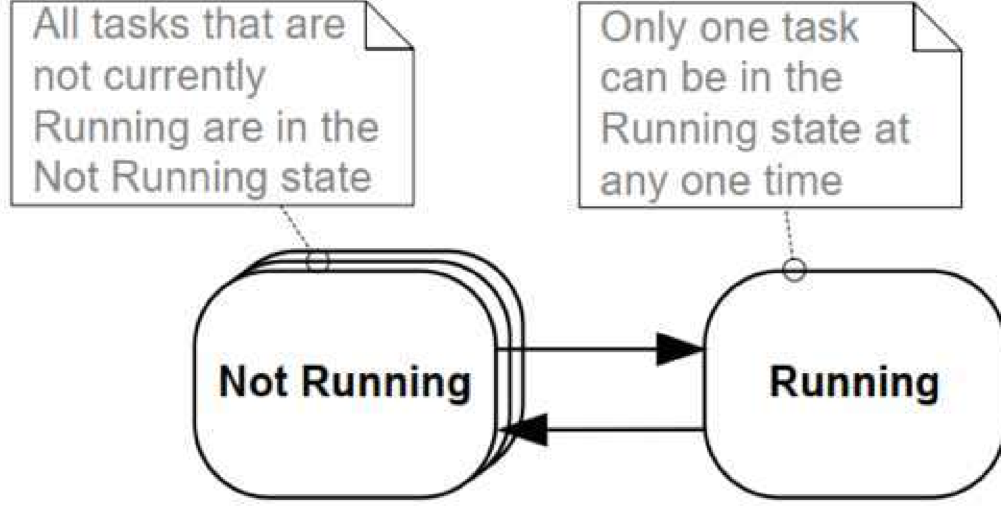
```
        /* Görev işlevselliğini uygulayacak kod buraya gelecektir. */

    }


    /*

     * Eğer görev uygulaması yukarıdaki döngüden çıkarsa, görev

     * kendisini uygulayan işlevin sonuna ulaşmadan önce silinmelidir.

     * vTaskDelete() API işlevine parametre olarak NULL iletildiğinde,

     * bu silinecek görevin çağıran (bu) görev olduğunu gösterir.

     */

    vTaskDelete( NULL );

}
```

Liste 4.2 Tipik bir görev işlevinin yapısı

4.3 En Üst Düzey Görev Durumları (Top Level Task States)

Bir uygulama birçok görevden oluşabilir. Uygulamayı çalıştıran işlemci tek bir çekirdek (core) içeriyorsa, herhangi bir anda yalnızca bir görev yürütülebilir. Bu, bir görevin iki durumdan birinde var olabileceği anlamına gelir: Çalışıyor (Running) ve Çalışmıyor (Not Running). Bu basitleştirilmiş model ilk olarak ele alınmıştır. Bu bölümün ilerleyen kısımlarında Çalışmıyor durumunun çeşitli alt durumları açıklanmaktadır.

İşlemci o görevin kodunu yürütürken bir görev Çalışıyor (Running) durumundadır. Bir görev Çalışmıyor (Not Running) durumundayken görev askıya alınmıştır ve çizgeleyici (scheduler) bir dahaki sefere Çalışıyor durumuna girmesi gerektiğine karar verdiğinde yürütülmeye devam edebilmesi için durumu (state) kaydedilmiştir. Bir görev yürütülmeye devam ettiğinde, Çalışıyor durumundan ayrılmadan önce yürütmek üzere olduğu talimattan devam eder.



Şekil 4.1 En üst düzey görev durumları ve geçişleri

Çalışmıyor durumundan Çalışıyor durumuna geçen bir görevin "değiştirildiği (switched in)" veya "takas edildiği (swapped in)" söylenir. Tersine, Çalışıyor durumundan Çalışmıyor durumuna geçen bir görevin "değiştirildiği (switched out)" veya "takas edildiği (swapped out)" söylenir. FreeRTOS zamanlayıcısı, bir görevi Çalışıyor durumuna sokup çıkarabilen tek varlıktır.

4.4 Görev Oluşturma (Task Creation)

Görevleri oluşturmak için altı API işlevi kullanılabilir: `xTaskCreate()`, `xTaskCreateStatic()`, `xTaskCreateRestricted()`, `xTaskCreateRestrictedStatic()`, `xTaskCreateAffinitySet()` ve `xTaskCreateStaticAffinitySet()`.

Her görev iki blok RAM gerektirir: biri Görev Kontrol Bloğunu (Task Control Block - `TCB`) tutmak için ve diğeri yığınını (stack) depolamak için. Adında "Static" bulunan FreeRTOS API işlevleri, işlevlere parametre olarak geçirilen önceden tahsis edilmiş RAM bloklarını kullanır. Tersine, adında "Static" bulunmayan API işlevleri gerekli RAM'i çalışma zamanında sistem yığınından (system heap) dinamik olarak ayırır.

Bazı FreeRTOS bağlantı noktaları (ports) "kısıtlı (restricted)" veya "ayrıcalıksız (unprivileged)" modda çalışan görevleri destekler. Adında "Restricted" bulunan FreeRTOS API işlevleri, sistemin belleğine sınırlı erişimle yürütülen görevler oluşturur. Adında "Restricted" bulunmayan API işlevleri, "ayrıcalıklı modda (privileged mode)" yürütülen ve sistemin tüm bellek haritasına erişimi olan görevler oluşturur.

Simetrik Çoklu İşlemeyi (Symmetric Multi Processing - SMP) destekleyen FreeRTOS bağlantı noktaları, farklı görevlerin aynı CPU'nun birden çok çekirdeğinde aynı anda çalışmasına olanak tanır. Bu bağlantı noktaları için, adında "Affinity" bulunan işlevleri kullanarak bir görevin hangi çekirdekte çalışacağını belirtebilirsiniz.

FreeRTOS görev oluşturma API işlevleri oldukça karmaşıktır. Bu belgedeki çoğu örnek, bu işlevlerin en basiti olduğu için `xTaskCreate()` işlevini kullanır.

4.4.1 xTaskCreate() API İşlevi

Liste 4.3 `xTaskCreate()` API işlev prototipini göstermektedir. `xTaskCreateStatic()` işlevinin sırasıyla görevin veri yapısını ve yığınını tutmak için önceden ayrılmış belleğe işaret eden iki ek parametresi vardır. Bölüm 2.5: Veri Türleri ve Kodlama Stili Kılavuzu, kullanılan veri türlerini ve adlandırma kurallarını açıklamaktadır.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,

                        const char * const pcName,

                        configSTACK_DEPTH_TYPE usStackDepth,

                        void * pvParameters,

                        UBaseType_t uxPriority,

                        TaskHandle_t * pxCreatedTask );
```

Liste 4.3 `xTaskCreate()` API işlev prototipi

`xTaskCreate()` Parametreleri ve dönüş değeri:

- **`pvTaskCode`:** Görevler basitçe asla çıkış yapmayan C işlevleridir ve bu nedenle normalde sonsuz bir döngü olarak uygulanırlar. `pvTaskCode` parametresi basitçe görevi uygulayan işleve giden bir işaretçidir (aslında sadece işlevin adıdır).
- **pcName:** Görev için açıklayıcı bir ad. FreeRTOS bunu hiçbir şekilde kullanmaz ve tamamen hata ayıklamaya yardımcı olması için dahil edilmiştir. Bir görevi insan tarafından okunabilen bir isimle tanımlamak, onu tanıtıcısıyla (handle) tanımlamaktan çok daha basittir. `configMAX_TASK_NAME_LEN` yapılandırma sabiti, `NULL` sonlandırıcı dahil olmak üzere bir görev adının olabileceği maksimum uzunluğu tanımlar.
- **`usStackDepth`:** Görev tarafından kullanılmak üzere tahsis edilecek yığının (stack) boyutunu belirtir. Dinamik olarak tahsis edilmiş bellek yerine önceden tahsis edilmiş belleği kullanmak için `xTaskCreate()` yerine `xTaskCreateStatic()` işlevini kullanın. Not: Değer bayt sayısını değil, yığının tutabileceği kelime (word) sayısını belirtir. Örneğin, yığın 32 bit genişliğindeyse ve `usStackDepth` 128 ise, `xTaskCreate()` 512 bayt yığın alanı ayırır (128 * 4 bayt).
- **`pvParameters`:** Görevleri uygulayan işlevler tek bir `void` işaretçi (`void` *) parametresi kabul eder. `pvParameters`, bu parametre kullanılarak göreve geçirilen değerdir.
- **`uxPriority`:** Görevin önceliğini tanımlar. 0 en düşük önceliktir ve (`configMAX_PRIORITIES` - 1) en yüksek önceliktir. Bölüm 4.5 `configMAX_PRIORITIES` sabitini açıklamaktadır.
- **`pxCreatedTask`:** Oluşturulan görevin tanıtıcısını (handle) saklayacak değişkenin adresi. Bu tanıtıcı, daha sonra görevin önceliğini değiştirmek veya

görevi silmek gibi API çağrılarında kullanılır. `pxCreatedTask` isteğe bağlıdır; tanıtıcıya ihtiyaç yoksa `NULL` verilebilir.

Dönüş değerleri:

- **`pdPASS`**: Bu, görevin başarıyla oluşturulduğunu gösterir.
- **`pdFAIL`**: Bu, görevi oluşturmak için yeterli yığın belleğinin bulunmadığını gösterir.

Örnek 4.1 Görevler oluşturma

Aşağıdaki örnek, iki basit görev oluşturmak ve ardından yeni oluşturulan görevleri başlatmak için gereken adımları göstermektedir. Görevler basitçe bir dizi periyodik gecikme oluşturmak için çok ilkel bir meşgul döngüsü (busy loop) kullanarak periyodik olarak bir dize yazdırır. Her iki görev de aynı öncelikte oluşturulur ve yazdırdıkları dize haricinde birbirinin aynısıdır - ilgili uygulamaları için Liste 4.4 ve Liste 4.5'e bakın.

```
void vTask1( void * pvParameters )

{

    /* ulCount'un optimize edilmediğinden emin olmak için volatile olarak
bildirilmiştir. */

    volatile unsigned long ulCount;



    for( ;; )

    {

        /* Geçerli görevin adını yazdırın. */

        vPrintLine( "Task 1 is running" );



        /* Bir süre gecikme. */

        for( ulCount = 0; ulCount < mainDELAY_LOOP_COUNT; ulCount++ )

        {

            /* Bu döngü sadece çok kaba bir gecikme uygulamasıdır.

               Burada yapılacak hiçbir şey yoktur. */

        }

    }

}
```

Liste 4.4 Örnek 4.1'de kullanılan ilk görevin uygulanması

```
void vTask2( void * pvParameters )

{

    /* ulCount'un optimize edilmediğinden emin olmak için volatile olarak
bildirilmiştir. */

    volatile unsigned long ulCount;



    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */

    for( ;; )

    {

        /* Bu görevin adını yazdırın. */

        vPrintLine( "Task 2 is running" );



        /* Bir süre gecikme. */

        for( ulCount = 0; ulCount < mainDELAY_LOOP_COUNT; ulCount++ )

        {

            /* Bu döngü sadece çok kaba bir gecikme uygulamasıdır. */

        }

    }

}
```

Liste 4.5 Örnek 4.1'de kullanılan ikinci görevin uygulanması

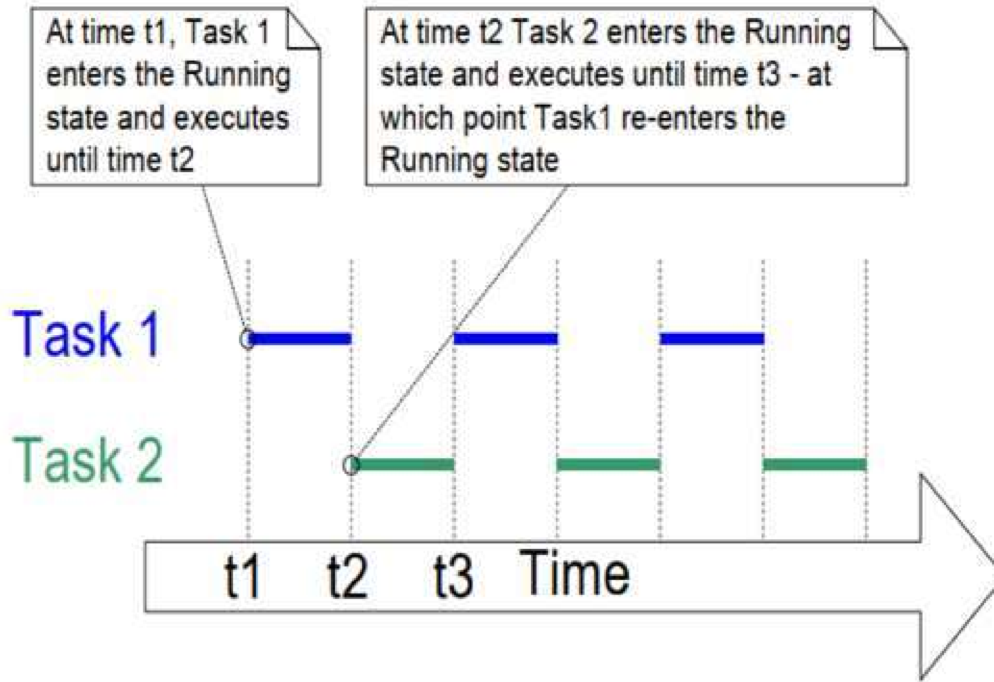
```
int main( void )

{

    /* İki görevden birini oluşturun. */

    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );



    /* Diğer görevi aynı şekilde ve aynı öncelikle oluşturun. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );
```

```
    /* Çizgeleyiciyi başlatın, böylece görevler yürütülmeye başlar. */
    vTaskStartScheduler();


    for( ;; );
}
```

Liste 4.6 Örnek 4.1 görevlerinin başlatılması

Şekil 4.2 Örnek 4.1 yürütüldüğünde üretilen çıktı

Şekil 4.2 her iki görevin de aynı anda çalışıyormuş gibi göründüğünü göstermektedir; ancak her iki görev de aynı işlemci çekirdeği üzerinde yürütüldüğü için durum böyle olamaz. Gerçekte her iki görev de Çalışıyor (Running) durumuna hızla girip çıkmaktadır. Her iki görev de aynı öncelikte çalışır ve bu nedenle aynı işlemci çekirdeğinde zamanı paylaşırlar. Şekil 4.3 gerçek yürütme modellerini göstermektedir.



Şekil 4.3 Örnek 4.1'deki iki görevin gerçek yürütme deseni

Örnek 4.1 her iki görevi de çizgeleyiciyi başlatmadan önce `main()` içinden oluşturdu. Bir görevi başka bir görevin içinden oluşturmak da mümkündür. Örneğin, Görev 2 Liste 4.7'de gösterildiği gibi Görev 1'in içinden oluşturulabilirdi.

```
void vTask1( void * pvParameters )

{

    const char *pcTaskName = "Task 1 is running\r\n";

    volatile unsigned long ul; /* ul'nin optimize edilmediğinden emin olmak için volatile. */



    /*

     * Eğer bu görev kodu yürütülüyorsa, çizgeleyici zaten başlatılmış demektir.

     * Sonsuz döngüye girmeden önce diğer görevi oluşturun.

     */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );



    for( ;; )

    {
```

```
        /* Bu görevin adını yazdırın. */

        vPrintLine( pcTaskName );


        /* Bir süre gecikme. */

        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {

            /* Bu döngü sadece çok kaba bir gecikme uygulamasıdır. */

        }

    }

}
```

Liste 4.7 Çizgeleyici başlatıldıktan sonra bir görevin içinden başka bir görev oluşturma

Örnek 4.2 Görev parametresini kullanma

Örnek 4.1'de oluşturulan iki görev neredeyse aynıdır, aralarındaki tek fark yazdırdıkları metin dizesidir. Tek bir görev uygulamasının iki örneğini oluşturursanız ve dizeyi her bir örneğe geçirmek için görev parametresini kullanırsanız, bu yinelemeyi (duplication) ortadan kaldırır.

Örnek 4.2, Örnek 4.1'de kullanılan iki görev işlevini, Liste 4.8'de gösterildiği gibi `vTaskFunction()` adlı tek bir görev işleviyle değiştirir. Görevin yazdırması gereken dizeyi elde etmek için görev parametresinin nasıl `char *` değerine dönüştürüldüğüne (cast edildiğine) dikkat edin.

```
void vTaskFunction( void * pvParameters )

{

    char *pcTaskName;

    volatile unsigned long ul; /* ul'nin optimize edilmediğinden emin olmak için
volatile. */



    /*

     * Yazdırılacak dize parametre aracılığıyla aktarılır. Bunu bir karakter
işaretçisine

     * dönüştürün (cast edin).

     */
```

```
    pcTaskName = ( char * ) pvParameters;

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */
    for( ;; )
    {
        /* Bu görevin adını yazdırın. */
        vPrintLine( pcTaskName );

        /* Bir süre gecikme. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* Bu döngü sadece çok kaba bir gecikme uygulamasıdır. */
        }
    }
}
```

Liste 4.8 Örnek 4.2'de iki görev oluşturmak için kullanılan tek görev işlevi

Liste 4.9, `vTaskFunction()` tarafından uygulanan görevin iki örneğini oluşturur ve görevin parametresini kullanarak her birine farklı bir dize geçirir. Her iki görev de FreeRTOS zamanlayıcısının kontrolü altında bağımsız olarak ve kendi yığınlarıyla (stack) yürütülür ve dolayısıyla `pcTaskName` ve `ul` değişkenlerinin kendi kopyalarıyla yürütülür.

```
/*
 * Görev parametreleri olarak aktarılacak dizeleri (strings) tanımlayın.
 * Görevler yürütülürken geçerli kalmalarını sağlamak için main() tarafından
kullanılan yığında
 * (stack) değil, const olarak tanımlanırlar.
 */
static const char * pcTextForTask1 = "Task 1 is running";
static const char * pcTextForTask2 = "Task 2 is running";
```

```
int main( void )

{

    /* İlk görevi oluştur. */

    xTaskCreate( vTaskFunction, /* Görevi uygulayan işleve giden işaretçi. */

                 "Task 1",      /* Görevin metin adı. Sadece hata ayıklama
içindir. */

                 1000,          /* Yığın derinliği (Stack depth). */

                 ( void * ) pcTextForTask1, /* Görev parametresini kullanarak
yazdırılacak metni aktarın. */

                 1,             /* Bu görev 1 önceliğinde (priority 1)
çalışacaktır. */

                 NULL );        /* Görev tanıtıcısı (handle) bu örnekte
kullanılmamıştır. */



    /*

     * Diğer görevi de tamamen aynı şekilde oluşturun. Bu kez AYNI görev
uygulamasından

     * (vTaskFunction) birden fazla görev oluşturulduğuna dikkat edin.

     * Sadece parametrede aktarılan değer farklıdır.

     * Aynı görev tanımının iki örneği oluşturulmaktadır.

     */

    xTaskCreate( vTaskFunction, "Task 2", 1000, ( void * ) pcTextForTask2, 1, NULL
);



    /* Çizgeleyiciyi başlatın, böylece görevler yürütülmeye başlar. */

    vTaskStartScheduler();



    for( ;; )

    {

    }

}
```

Liste 4.9 Örnek 4.2 için `main()` işlevi

Örnek 4.2'nin çıktısı tamamen Şekil 4.2'de Örnek 4.1 için gösterilenle aynıdır.

4.5 Görev Öncelikleri (Task Priorities)

FreeRTOS zamanlayıcısı her zaman çalışabilecek en yüksek öncelikli (highest priority) görevin Çalışıyor (Running) durumuna girmek üzere seçilen görev olmasını sağlar. Eşit önceliğe sahip görevler sırayla Çalışıyor durumuna geçirilir ve bu durumdan çıkarılır.

Görevi oluşturmak için kullanılan API işlevinin `uxPriority` parametresi, göreve ilk önceliğini verir. `vTaskPrioritySet()` API işlevi, bir görevin önceliğini oluşturulduktan sonra değiştirir.

Uygulama tanımlı `configMAX_PRIORITIES` derleme zamanı yapılandırma sabiti, kullanılabilir önceliklerin sayısını ayarlar. Düşük sayısal öncelik değerleri düşük öncelikli görevleri ifade eder ve 0 önceliği mümkün olan en düşük önceliktir — dolayısıyla geçerli öncelikler 0 ile (`configMAX_PRIORITIES` - 1) arasında değişir. İstenilen sayıda görev aynı önceliği paylaşabilir.

FreeRTOS zamanlayıcısı, Çalışıyor durumundaki görevi seçmek için kullanılan algoritmanın iki uygulamasına (implementation) sahiptir ve `configMAX_PRIORITIES` için izin verilen maksimum değer kullanılan uygulamaya bağlıdır:

4.5.1 Genel Çizgeleyici (Generic Scheduler)

Genel çizgeleyici C dilinde yazılmıştır ve tüm FreeRTOS mimari bağlantı noktalarıyla (ports) kullanılabilir. `configMAX_PRIORITIES` üzerinde bir üst sınır (upper limit) dayatmaz. Genel olarak, `configMAX_PRIORITIES` değerinin en aza indirilmesi tavsiye edilir, çünkü daha fazla değer daha fazla RAM gerektirir ve en kötü durum yürütme süresinin (worst-case execution time) daha uzun olmasına yol açar.

4.5.2 Mimari Olarak Optimize Edilmiş Çizgeleyici (Architecture-Optimized Scheduler)

Mimari olarak optimize edilmiş uygulamalar, mimariye özgü assembly kodunda yazılır ve genel C uygulamasından daha performanslıdır ve en kötü durum yürütme süresi tüm `configMAX_PRIORITIES` değerleri için aynıdır.

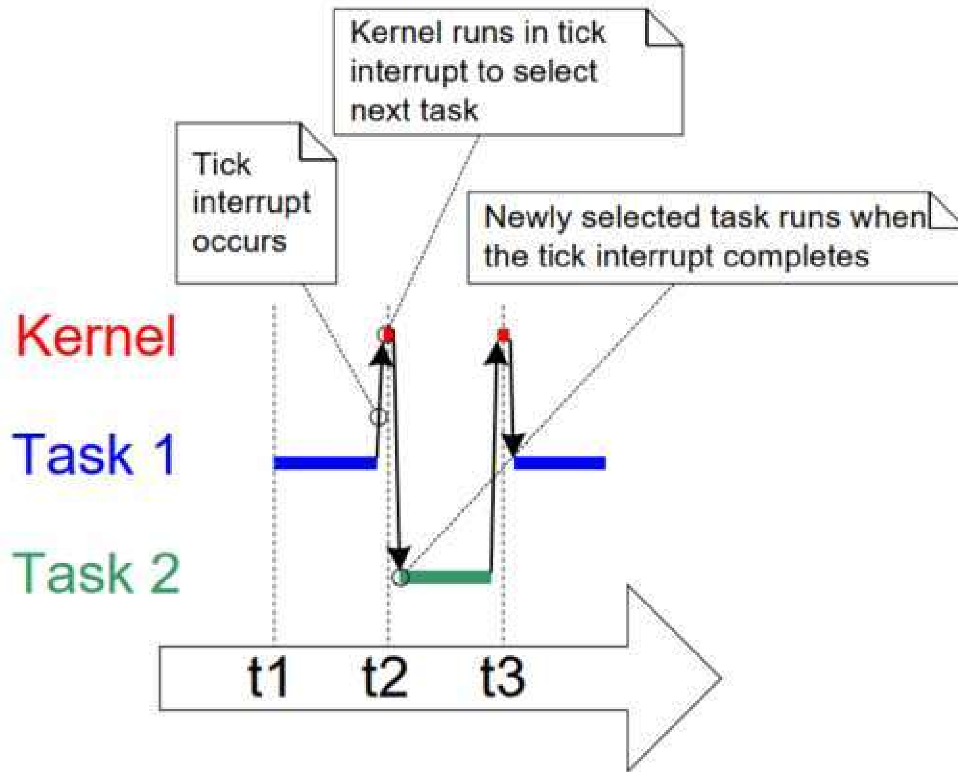
Mimari optimize edilmiş uygulama, 32 bit mimarilerde 32 ve 64 bit mimarilerde 64 olmak üzere `configMAX_PRIORITIES` için maksimum bir değer dayatır (impose). Genel yöntemde olduğu gibi, daha yüksek değerler daha fazla RAM gerektirdiğinden `configMAX_PRIORITIES`'i pratik en düşük değerde tutmanız tavsiye edilir.

Mimari optimize edilmiş uygulamayı kullanmak için `FreeRTOSConfig.h`'de `configUSE_PORT_OPTIMIZED_TASK_SELECTION` değerini 1 veya genel uygulamayı kullanmak için 0 olarak ayarlayın. Tüm FreeRTOS bağlantı noktaları mimari olarak optimize edilmiş bir uygulamaya sahip değildir. Sahip olanlar, tanımsız bırakılırsa `configUSE_PORT_OPTIMIZED_TASK_SELECTION` değerini varsayılan olarak 1 yapar. Olmayanlar ise 0'a varsayılan olarak ayarlar.

4.6 Zaman Ölçümü ve Tick Kesmesi (Tick Interrupt)

Bölüm 4.12, Çizgeleme Algoritmaları, 'zaman dilimleme' (time slicing) adı verilen isteğe bağlı bir özelliği açıklamaktadır. Zaman dilimleme şimdiye kadar sunulan örneklerde kullanılmıştır ve ürettikleri çıktıda gözlemlenen davranıştır. Örneklerde, her iki görev de aynı öncelikte oluşturulmuştur ve her iki görev de her zaman çalışabilmektedir. Bu nedenle, her görev bir 'zaman dilimi' boyunca yürütülmüş, bir zaman diliminin başlangıcında Çalışıyor durumuna girmiş ve bir zaman diliminin sonunda Çalışıyor durumundan çıkmıştır. Şekil 4.3'te t1 ve t2 arasındaki süre tek bir zaman dilimine eşittir.

Çizgeleyici (scheduler), çalıştırılacak bir sonraki görevi seçmek için her zaman diliminin sonunda yürütülür. Bu amaçla 'tick interrupt' (tick kesmesi) adı verilen periyodik bir kesme kullanılır. `configTICK_RATE_HZ` derleme zamanı yapılandırma sabiti tick kesmesinin frekansını ve dolayısıyla her bir zaman diliminin uzunluğunu belirler. Örneğin, `configTICK_RATE_HZ` değerinin 100 (Hz) olarak ayarlanması her bir zaman diliminin 10 milisaniye sürmesiyle sonuçlanır. İki tick kesmesi arasındaki süreye 'tick period' denir — yani bir zaman dilimi bir tick periyoduna eşittir.



Şekil 4.4 Tick kesmesinin yürütülmesini göstermek için genişletilmiş yürütme dizisi

`configTICK_RATE_HZ` için optimal değer uygulamaya bağlıdır, ancak 100 değeri tipiktir.

FreeRTOS API çağrıları zamanı tick periyotlarının katları cinsinden belirtir ve buna genellikle 'tick' (kene/tık) denir. `pdMS_TO_TICKS()` makrosu, milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürür. Kullanılabilen çözünürlük tanımlanan kene frekansına bağlıdır ve kene frekansı 1KHz'in üzerindeyse (`configTICK_RATE_HZ` 1000'den büyükse) `pdMS_TO_TICKS()` kullanılamaz. Liste 4.10, 200 milisaniye olarak belirtilen bir süreyi tick periyotlarında belirtilen eşdeğer bir süreye dönüştürmek için `pdMS_TO_TICKS()` öğesinin nasıl kullanılacağını gösterir.

```
/*

 * pdMS_TO_TICKS() tek parametresi olarak milisaniye cinsinden bir zaman alır,

 * ve tick periyotlarındaki eşdeğer zamanı değerlendirir.

 * Bu örnekte xTimeInTicks değişkeninin 200 milisaniyeye eşdeğer olan

 * tick periyodu sayısına ayarlanması gösterilmektedir.

 */

TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

Liste 4.10 200 milisaniyeyi tick periyotlarında eşdeğer bir süreye dönüştürmek için `pdMS_TO_TICKS()` makrosunun kullanılması

Zamanları doğrudan tick olarak değil de milisaniye olarak belirtmek için `pdMS_TO_TICKS()` kullanmak, uygulama içinde belirtilen zamanların tick frekansı değiştirilirse değişmemesini sağlar.

'Tick count' (tick sayısı), tick sayısının taşmadığı (overflow yapmadığı) varsayılarak, çizgeleyici başladığından beri meydana gelen tick kesmelerinin toplam sayısıdır. FreeRTOS zaman tutarlılığını dahili olarak yönettiğinden, gecikme sürelerini belirtirken kullanıcı uygulamalarının taşmaları dikkate alması gerekmez.

Bölüm 4.12: Çizgeleme Algoritmaları, çizgeleyicinin çalıştırılacak yeni bir görevi ne zaman seçeceğini ve tick kesmesinin ne zaman yürütüleceğini etkileyen yapılandırma sabitlerini (configuration constants) açıklamaktadır.

Örnek 4.3 Önceliklerle (priorities) denemeler yapmak

Çizgeleyici her zaman çalışabilecek en yüksek öncelikli görevin Çalışıyor (Running) durumuna girmek üzere seçilen görev olmasını sağlayacaktır. Şimdiye kadarki örneklerde aynı öncelikte iki görev yaratıldı, bu nedenle her ikisi de sırayla Çalışıyor durumuna girdi ve çıktı. Bu örnek, görevlerin öncelikleri farklı olduğunda neler olacağına bakmaktadır. Liste 4.11, ilki 1 önceliğine ve ikincisi 2 önceliğine sahip görevleri oluşturmak için kullanılan kodu göstermektedir. Her iki görevi de uygulayan tek işlev değişmemiştir; bir gecikme yaratmak için bir boş döngü (null loop) kullanarak periyodik olarak bir dize yazdırır.

```
/* Görev parametreleri olarak aktarılacak dizeleri tanımlayın. */
```

```
static const char * pcTextForTask1 = "Task 1 is running";

static const char * pcTextForTask2 = "Task 2 is running";


int main( void )

{

    /* İlk görevi 1 önceliğiyle oluşturun. */

    xTaskCreate( vTaskFunction, "Task 1", 1000, ( void * ) pcTextForTask1, 1, NULL
);


    /* İkinci görevi daha yüksek bir öncelik olan 2 ile oluşturun. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, ( void * ) pcTextForTask2, 2, NULL
);


    /* Çizgeleyiciyi başlatın, böylece görevler yürütülmeye başlar. */

    vTaskStartScheduler();


    /* Buraya ulaşmayacak. */

    return 0;

}
```

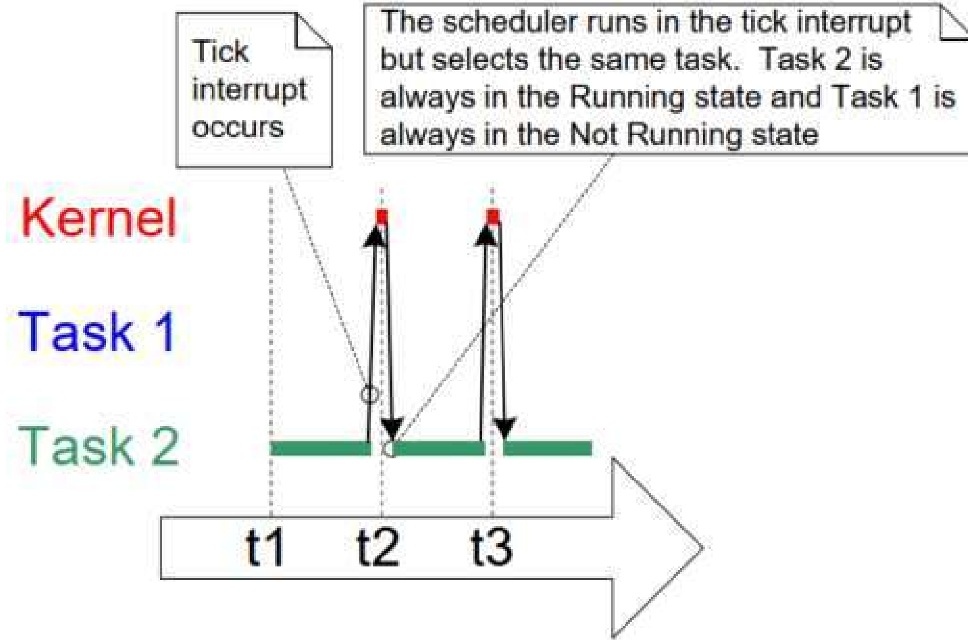
Liste 4.11 Farklı önceliklerde iki görev oluşturma

```
C:\Temp>rtosdemo
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```

Şekil 4.5 Her iki görevi farklı önceliklerde çalıştırma

Çizgeleyici her zaman çalışabilecek en yüksek öncelikli görevi seçecektir. Görev 2, Görev 1'den daha yüksek bir önceliğe sahiptir ve her zaman çalışabilir; bu nedenle çizgeleyici her zaman Görev 2'yi seçer ve Görev 1 asla yürütülmez. Görev 1'in işlem süresinin Görev 2 tarafından 'aç bırakıldığı' (starved) söylenir - asla Çalışıyor durumunda olmadığı için dizesini yazdıramaz.

Görev 2 her zaman çalışabilir, çünkü hiçbir zaman hiçbir şeyi beklemesi gerekmez - ya boş bir döngüde dönüyor ya da terminale yazdırıyordur.



Şekil 4.6 Örnek 4.3'ten bir görevin diğerinden daha yüksek önceliğe sahip olduğu zamanki yürütme deseni

4.7 Çalışmıyor (Not Running) Durumunu Genişletme

Şimdiye kadar, oluşturulan görevler her zaman gerçekleştirecek işlemlere sahipti ve asla hiçbir şey için beklemek zorunda kalmadılar ve hiçbir şey için beklemek zorunda olmadıkları için her zaman Çalışıyor (Running) durumuna girebildiler. Bu tür 'sürekli işleyen' (continuous processing) görevlerin yararlılığı sınırlıdır çünkü yalnızca en düşük öncelikte oluşturulabilirler. Başka bir öncelikte çalışırlarsa, daha düşük öncelikli görevlerin çalışmasını engellerler.

Bu görevleri faydalı kılmak için, olay güdümlü (event-driven) olacak şekilde yeniden yazılmalıdırlar. Olay güdümlü bir görevin yalnızca bir olay onu tetikledikten sonra yapacak işi (işlemi) vardır ve bu zamandan önce Çalışıyor durumuna giremez. Çizgeleyici her zaman çalışabilen en yüksek öncelikli görevi seçer. Yüksek öncelikli bir görev bir olay beklediği için seçilemezse, çizgeleyici bunun yerine çalışabilen daha

düşük öncelikli bir görevi seçmelidir. Bu nedenle, olay güdümlü görevler yazmak, en yüksek öncelikli görevlerin tüm düşük öncelikli görevleri işlem süresinden mahrum bırakmadan, görevlerin farklı önceliklerde oluşturulabileceği anlamına gelir.

4.7.1 Engellenmiş (Blocked) Durumu

Bir olayı bekleyen bir görevin, Çalışmıyor durumunun bir alt durumu olan 'Engellendi' veya 'Engellenmiş' (Blocked) durumunda olduğu söylenir.

Görevler iki farklı olay türünü beklemek için Engellenmiş durumuna girebilir:

1. **Zamansal (zamanla ilgili) olaylar:** Bu olaylar bir gecikme süresi sona erdiğinde veya mutlak bir zamana ulaşıldığında meydana gelir. Örneğin, bir görev 10 milisaniyenin geçmesini beklemek için Engellenmiş durumuna girebilir.
2. **Senkronizasyon (eşzamanlama) olayları:** Bu olaylar başka bir görev veya kesmeden (interrupt) kaynaklanır. Örneğin, bir görev bir kuyruğa veri gelmesini beklemek için Engellenmiş durumuna girebilir. Senkronizasyon olayları çok çeşitli olay türlerini kapsar.

FreeRTOS kuyrukları, ikili semaforlar, sayım semaforları, muteksler, özyinelemeli muteksler, olay grupları, akış arabellekleri, mesaj arabellekleri ve göreve doğrudan bildirimler senkronizasyon olayları oluşturabilir. Sonraki bölümlerde bu özelliklerin çoğu ele alınacaktır.

Bir görev, bir senkronizasyon olayı üzerinde bir zaman aşımı (timeout) ile engellenebilir ve etkili bir şekilde her iki olay türünde de aynı anda engellenebilir. Örneğin, bir görev kuyruğa veri gelmesi için en fazla 10 milisaniye beklemeyi seçebilir. Görev, 10 milisaniye içinde veri gelirse veya veri gelmeden 10 milisaniye geçerse Engellenmiş durumundan çıkar.

4.7.2 Askıya Alınmış (Suspended) Durumu

Askıya alındı da (Suspended) Çalışmıyor'un bir alt durumudur. Askıya Alınmış durumundaki görevler çizgeleyici tarafından kullanılamaz. Askıya Alınmış durumuna girmenin tek yolu `vTaskSuspend()` API işlevini çağırmaktır ve çıkmanın tek yolu `vTaskResume()` veya `xTaskResumeFromISR()` API işlevlerini çağırmaktır. Çoğu uygulama Askıya Alınmış durumunu kullanmaz.

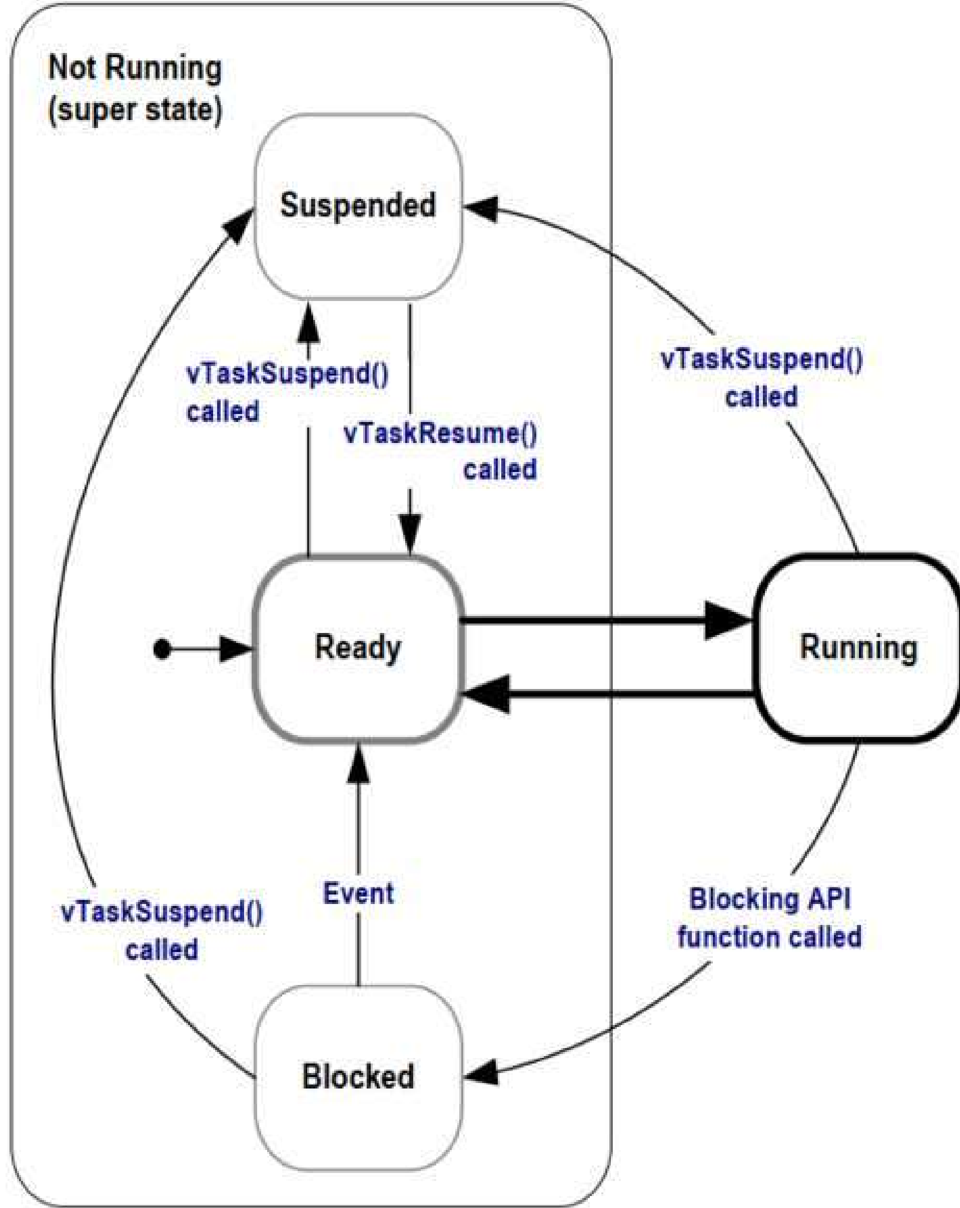
4.7.3 Hazır (Ready) Durumu

Çalışmıyor durumunda olan ve Engellenmiş Olmayan veya Askıya Alınmamış görevlerin Hazır (Ready) durumunda olduğu söylenir. Çalışabilirler ve bu nedenle çalışmaya 'hazırdırlar', ancak şu anda Çalışıyor (Running) durumunda değildirler.

4.7.4 Durum Geçiş Diyagramını Tamamlama

Şekil 4.7, bu bölümde açıklanan tüm Çalışmıyor alt durumlarını içerecek şekilde basitleştirilmiş durum diyagramını genişletmektedir. Şimdiye kadar örneklerde

oluşturulan görevler Engellenmiş veya Askıya Alınmış durumlarını kullanmamıştır. Sadece Şekil 4.7'de kalın çizgilerle gösterildiği gibi Hazır durumu ile Çalışıyor durumu arasında geçiş yapmışlardır.



Şekil 4.7 Tam görev durum makinesi (Full task state machine)

Örnek 4.4 Bir gecikme (delay) oluşturmak için Engellenmiş durumunu kullanma

Şimdiye kadar sunulan örneklerde oluşturulan tüm görevler 'periyodik' olmuştur - bir süre gecikmişler ve sonra dizelerini yazdırmışlar, sonra tekrar gecikmişlerdir ve bu böyle devam etmiştir. Gecikme, bir boş döngü (null loop) kullanılarak çok kaba bir şekilde üretilmiştir - görev, sabit bir değere ulaşana kadar artan bir döngü sayacını

yoklamıştır (polled). Örnek 4.3 bu yöntemin dezavantajını açıkça göstermiştir. Daha yüksek öncelikli görev boş döngüyü yürütürken Çalışıyor durumunda kalmış ve daha düşük öncelikli görevi işlem süresinden 'aç bırakmıştır' (starved).

Verimsizliği de dahil olmak üzere her türlü yoklamanın (polling) başka birçok dezavantajı vardır. Yoklama (polling) sırasında, görevin aslında yapacak bir işi yoktur, ancak yine de maksimum işlem süresini kullanır ve böylece işlemci döngülerini (processor cycles) boşa harcar. Örnek 4.4, yoklama boş döngüsünü, prototipi Liste 4.12'de gösterilen `vTaskDelay()` API işlevine yapılan bir çağrıyla değiştirerek bu davranışı düzeltir. Yeni görev tanımı Liste 4.13'te gösterilmiştir. `vTaskDelay()` API işlevinin yalnızca `FreeRTOSConfig.h` içinde `INCLUDE_vTaskDelay` 1 olarak ayarlandığında kullanılabildiğine dikkat edin.

`vTaskDelay()`, çağıran görevi sabit sayıda tick kesmesi için Engellenmiş durumuna (Blocked state) yerleştirir. Görev, Engellenmiş durumundayken herhangi bir işlem süresi kullanmaz, bu nedenle görev yalnızca gerçekten yapılması gereken bir iş olduğunda işlem süresini kullanır.

```
void vTaskDelay( TickType_t xTicksToDelay );
```

Liste 4.12 `vTaskDelay()` API işlev prototipi

`vTaskDelay` parametreleri:

- **xTicksToDelay:** Çağıran görevin tekrar Hazır (Ready) durumuna geçirilmeden önce Engellenmiş durumunda kalacağı tick kesmelerinin sayısı. Örneğin, tick sayısı 10.000 iken bir görev `vTaskDelay( 100 )` çağrısı yaparsa, derhal Engellenmiş durumuna girer ve tick sayısı 10.100'e ulaşana kadar Engellenmiş durumunda kalır. Milisaniye cinsinden belirtilen bir süreyi tick cinsinden belirtilen bir süreye dönüştürmek için `pdMS_TO_TICKS()` makrosu kullanılabilir. Örneğin, `vTaskDelay( pdMS_TO_TICKS( 100 ) )` çağrısı, çağıran görevin 100 milisaniye boyunca Engellenmiş durumunda kalmasına neden olur.

```
void vTaskFunction( void * pvParameters )

{

    char * pcTaskName;

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );



    /*

     * Yazdırılacak dize parametre aracılığıyla aktarılır.

     * Bunu bir karakter işaretçisine dönüştürün (cast edin).

     */
```

```
    pcTaskName = ( char * ) pvParameters;

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */
    for( ;; )
    {
        /* Bu görevin adını yazdırın. */
        vPrintLine( pcTaskName );

        /*
         * Bir süre gecikme. Bu kez, gecikme süresi sona erene kadar görevi
         * Engellenmiş durumuna yerleştiren vTaskDelay() işlevine bir çağrı
kullanılır.
         * Parametre 'ticks' olarak belirtilen bir zaman alır ve
         * pdMS_TO_TICKS() makrosu 250 milisaniyeyi 'ticks' cinsinden eşdeğer bir
         * zamana dönüştürmek için kullanılır.
         */
        vTaskDelay( xDelay250ms );
    }
}
```

Liste 4.13 Boş döngü gecikmesini `vTaskDelay()` çağrısıyla değiştirdikten sonra örnek görev için kaynak kodu

İki görev hala farklı önceliklerde oluşturulmasına rağmen, artık her ikisi de çalışacaktır. Şekil 4.8'de gösterilen Örnek 4.4'ün çıktısı beklenen davranışı doğrulamaktadır.

```
C:\Temp>rtosdemo
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
```

Şekil 4.8 Örnek 4.4 yürütüldüğünde üretilen çıktı

Şekil 4.9'da gösterilen yürütme dizisi (execution sequence), farklı önceliklerde oluşturulmuş olsalar bile her iki görevin neden çalıştığını açıklar. Çizgeleyicinin (scheduler) kendisinin yürütülmesi basitlik için atlanmıştır.

Boş (idle) görev, her zaman çalışabilecek en az bir görev (Hazır durumunda en az bir görev) olmasını sağlamak için çizgeleyici başlatıldığında otomatik olarak oluşturulur. Bölüm 4.8: Boş Görev ve Boş Görev Kancası (The Idle Task and the Idle Task Hook) Boş görevi daha ayrıntılı olarak açıklamaktadır.

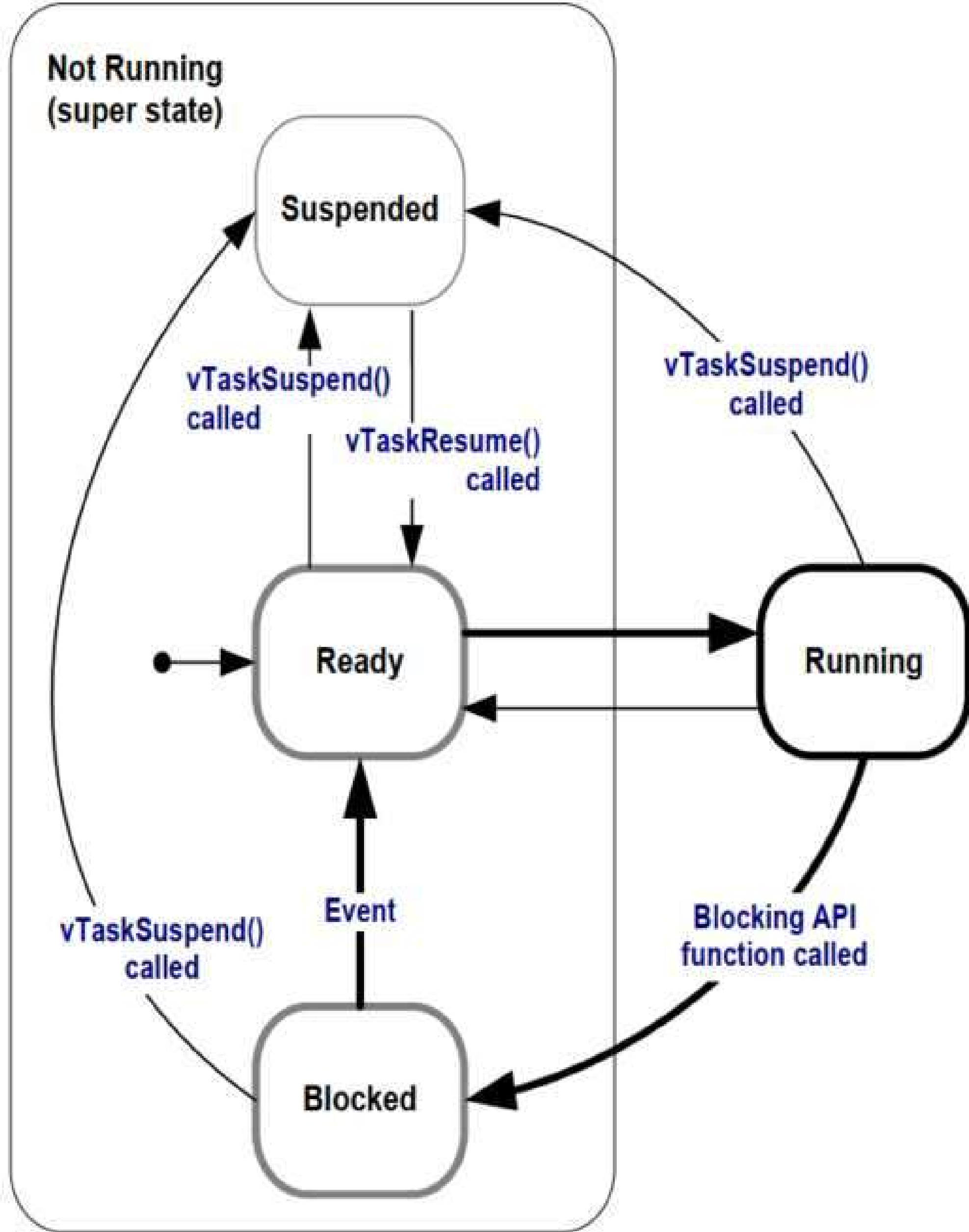
Şekil 4.9 Görevler boş döngü (null loop) yerine `vTaskDelay()` kullandığındaki yürütme sırası

Görevlerin yalnızca uygulaması (implementation) değişmiştir, işlevsellikleri değişmemiştir. Şekil 4.9'u Şekil 4.4 ile karşılaştırmak, bu işlevselliğin çok daha verimli bir şekilde başarıldığını açıkça göstermektedir.

Şekil 4.4, görevler gecikme oluşturmak için bir boş döngü kullandıklarında ve dolayısıyla her zaman çalışabildiklerindeki yürütme modelini gösterir. Sonuç olarak, aralarındaki kullanılabilir işlemci zamanının (processor time) yüzde yüzünü kullanırlar. Şekil 4.9, görevler gecikme sürelerinin tamamı boyunca Engellenmiş durumuna girdiklerinde yürütme modelini gösterir. Yalnızca gerçekten gerçekleştirilmesi gereken işleri (bu durumda basitçe yazdırılacak bir mesaj) olduğunda işlemci zamanını kullanırlar ve sonuç olarak kullanılabilir işlem süresinin yalnızca küçük bir kısmını kullanırlar.

Şekil 4.9'da gösterilen senaryoda, görevler Engellenmiş durumundan her çıktıklarında, Engellenmiş durumuna yeniden girmeden önce bir tick periyodunun bir bölümü boyunca çalışırlar. Çoğu zaman çalışabilecek (Hazır durumunda olan) hiçbir uygulama görevi yoktur ve bu nedenle Çalışıyor durumuna girmek üzere seçilebilecek hiçbir uygulama görevi yoktur. Bu durumda, boş (idle) görev çalışır. Boş göreve tahsis edilen işlem süresi miktarı, sistemdeki yedek işlem kapasitesinin bir ölçüsüdür. Bir RTOS kullanmak, yalnızca bir uygulamanın tamamen olay güdümlü (event driven) olmasına izin vererek yedek işlem kapasitesini önemli ölçüde artırabilir.

Şekil 4.10'daki kalın çizgiler Örnek 4.4'teki görevler tarafından gerçekleştirilen geçişleri (transitions) gösterir; her görev artık Hazır durumuna döndürülmeden önce Engellenmiş durumu üzerinden geçiş yapmaktadır.



Şekil 4.10 Kalın çizgiler, Örnek 4.4'teki görevler tarafından gerçekleştirilen durum geçişlerini gösterir

4.7.5 vTaskDelayUntil() API İşlevi

`vTaskDelayUntil()`, `vTaskDelay()` işlevine benzer. Az önce gösterildiği gibi, `vTaskDelay()` parametresi, bir görevin `vTaskDelay()` çağrısı yapması ile aynı görevin bir kez daha Engellenmiş durumundan çıkması arasında meydana gelmesi gereken tick kesmelerinin sayısını belirtir. Görevin Engellenmiş durumunda kaldığı süre `vTaskDelay()` parametresi ile belirtilir, ancak görevin Engellenmiş durumundan çıktığı zaman `vTaskDelay()`'in çağrıldığı zamana göre görelidir (relative).

Buna karşılık `vTaskDelayUntil()` işlevinin parametreleri, çağıran görevin Engellenmiş durumundan Hazır durumuna ne zaman (tam tick sayısı değeri olarak) geçirilmesi gerektiğini belirtir. Çağıran görevin engelinin kaldırıldığı (unblocked) zaman, işlevin çağrıldığı zamana göreceli (`vTaskDelay()` işlevinde olduğu gibi) olmak yerine mutlak (absolute) olduğundan, sabit bir yürütme periyodu (görevinizin sabit bir frekansta periyodik olarak yürütülmesini istediğiniz durumlar) gerektiğinde kullanılacak API işlevi `vTaskDelayUntil()`'dir.

```
void vTaskDelayUntil( TickType_t * pxPreviousWakeTime,

                      TickType_t xTimeIncrement );
```

Liste 4.14 `vTaskDelayUntil()` API işlev prototipi

`vTaskDelayUntil()` parametreleri:

- **`pxPreviousWakeTime`:** Bu parametre, `vTaskDelayUntil()` öğesinin düzenli (periyodik) ve sabit bir frekansta yürütülen bir görevi uygulamak için kullanıldığı varsayımıyla adlandırılmıştır. Bu durumda, `pxPreviousWakeTime` görevin Engellenmiş durumundan en son çıktığı (uyandırıldığı - woken up) zamanı tutar. Bu zaman, görevin Engellenmiş durumundan bir sonraki çıkış zamanını hesaplamak için bir referans noktası olarak kullanılır. `pxPreviousWakeTime` tarafından işaret edilen değişken `vTaskDelayUntil()` işlevinde otomatik olarak güncellenir; normalde uygulama kodu tarafından değiştirilmez, ancak ilk kullanımından önce geçerli tick sayısı (current tick count) ile ilklendirilmesi (initialized) gerekir. Liste 4.15 değişkenin nasıl ilklendirileceğini göstermektedir.
- **`xTimeIncrement`:** Bu parametre ayrıca `vTaskDelayUntil()` öğesinin periyodik olarak ve `xTimeIncrement` değeri tarafından ayarlanan sabit bir frekansta yürütülen bir görevi uygulamak için kullanıldığı varsayımıyla adlandırılmıştır. `xTimeIncrement` 'ticks' cinsinden belirtilir. Milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için `pdMS_TO_TICKS()` makrosu kullanılabilir.

Örnek 4.5 Örnek görevlerin `vTaskDelayUntil()` kullanacak şekilde dönüştürülmesi

Örnek 4.4'te oluşturulan iki görev periyodik görevlerdir, ancak `vTaskDelay()` kullanmak bunların çalışma frekansının (sıklığının) sabit olduğunu garanti etmez, çünkü görevlerin Engellenmiş durumundan ayrılma zamanı `vTaskDelay()` işlevini çağırdıkları zamana göredir (relative). Görevleri `vTaskDelay()` yerine `vTaskDelayUntil()` kullanacak şekilde dönüştürmek bu potansiyel sorunu çözer.

```
void vTaskFunction( void * pvParameters )

{

    char * pcTaskName;
```

```
    TickType_t xLastWakeTime;

    /* Yazdırılacak dize parametre aracılığıyla aktarılır. */
    pcTaskName = ( char * ) pvParameters;

    /*
     * xLastWakeTime değişkeninin geçerli kene (tick) sayısıyla başlatılması
     * gerekir. Bu değişkenin açıkça yazıldığı (explicitly written to) tek
     * zaman olduğuna dikkat edin. Bundan sonra xLastWakeTime,
     * vTaskDelayUntil() içinde otomatik olarak güncellenir.
     */
    xLastWakeTime = xTaskGetTickCount();

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */
    for( ;; )
    {
        /* Bu görevin adını yazdırın. */
        vPrintLine( pcTaskName );

        /*
         * Bu görev tam olarak her 250 milisaniyede bir yürütülmelidir.
         * vTaskDelay() işlevinde olduğu gibi, zaman tick (kene) cinsinden
         * ölçülür ve milisaniyeleri kenelere dönüştürmek için pdMS_TO_TICKS()
         * makrosu kullanılır. xLastWakeTime, vTaskDelayUntil() içinde otomatik
         * olarak güncellenir, bu nedenle görev tarafından açıkça güncellenmez.
         */
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    }
}
```

Liste 4.15 `vTaskDelayUntil()` kullanan örnek görevin uygulanması

Örnek 4.5 tarafından üretilen çıktı, tam olarak Şekil 4.8'de Örnek 4.4 için gösterildiği gibidir.

Örnek 4.6 Engelleyen (blocking) ve engellemeyen (non-blocking) görevleri birleştirme

Önceki örneklerde hem yoklama (polling) yapan hem de engelleyen (blocking) görevlerin davranışları izole olarak incelenmiştir. Bu örnek, beklenen sistem davranışına ilişkin daha önce söylediklerimizi pekiştirmekte ve bu iki yöntem birleştirildiğinde oluşan yürütme dizisini aşağıdaki gibi göstermektedir:

1. Öncelik 1'de iki görev oluşturulur. Bunlar sürekli olarak bir dize yazdırmaktan başka bir şey yapmazlar. Bu görevler asla Engellenmiş durumuna girmelerine neden olabilecek API işlev çağrıları yapmazlar, bu nedenle her zaman Hazır veya Çalışıyor durumundadırlar. Bu tür görevler 'sürekli işleme' (continuous processing) görevleri olarak adlandırılır, çünkü (bu durumda oldukça önemsiz işler olsa da) her zaman yapacak işleri vardır. Liste 4.16 sürekli işleme görevlerinin kaynak kodunu göstermektedir.
2. Daha sonra diğer iki görevin önceliğinin üzerinde olan öncelik 2'de üçüncü bir görev oluşturulur. Üçüncü görev de yalnızca bir dize yazdırır, ancak bu kez periyodik olarak, bu nedenle her yazdırma yinelemesi arasında kendisini Engellenmiş durumuna yerleştirmek için `vTaskDelayUntil()` API işlevini kullanır. Liste 4.17 periyodik görevin kaynak kodunu göstermektedir.

```
void vContinuousProcessingTask( void * pvParameters )
{
    char * pcTaskName;

    /* Yazdırılacak dize parametre aracılığıyla aktarılır. */
    pcTaskName = ( char * ) pvParameters;

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */
    for( ;; )
    {
        /*
         * Bu görevin adını yazdırın. Bu görev bunu hiçbir zaman
         * engellenmeden veya gecikmeden tekrar tekrar yapar.
         */
```

```
        vPrintLine( pcTaskName );

    }

}
```

Liste 4.16 Örnek 4.6'da kullanılan sürekli işleme görevi

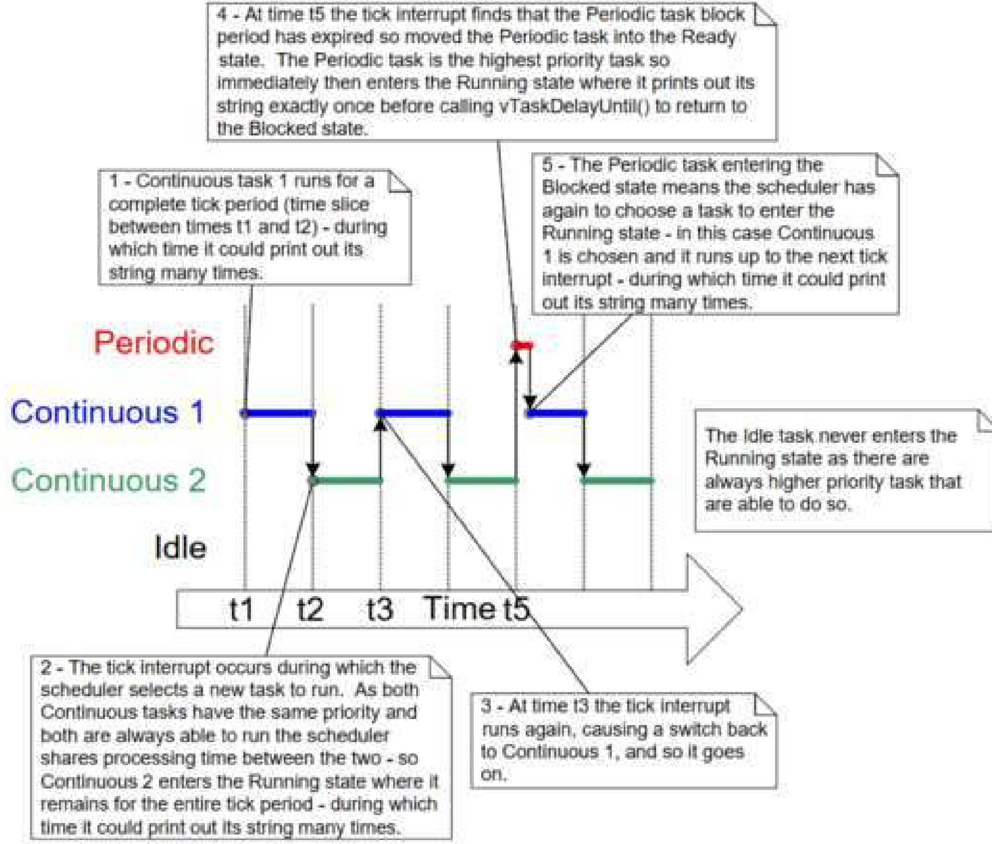
```
void vPeriodicTask( void * pvParameters )

{

    TickType_t xLastWakeTime;

    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );


    /* xLastWakeTime değişkeninin geçerli tick sayısı ile başlatılması gerekir. */

    xLastWakeTime = xTaskGetTickCount();


    for( ;; )

    {

        /* Bu görevin adını yazdırın. */

        vPrintLine( "Periodic task is running" );


        /*

         * Görev tam olarak her 3 milisaniyede bir yürütülmelidir -

         * bu işlevdeki xDelay3ms bildirimine bakın.

         */

        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );

    }

}
```

Liste 4.17 Örnek 4.6'da kullanılan periyodik görev

```
Continuous task 2 running
Continuous task 2 running
Periodic task is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 2 is running
Continuous task 2 is running
Continuous task 2 is running
Continuous task 2 is running
Continuous task 2 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Continuous task 1 is running
Periodic task is running
Continuous task 2 running
Continuous task 2 running
```

Şekil 4.11 Örnek 4.6 yürütüldüğünde üretilen çıktı

Şekil 4.11 Örnek 4.6 tarafından üretilen çıktıyı gösterirken, gözlemlenen davranışın açıklaması Şekil 4.12'de gösterilen yürütme dizisi tarafından verilmektedir.



Şekil 4.12 Örnek 4.6'nın yürütme deseni

4.8 Boş Görev ve Boş Görev Kancası (The Idle Task and the Idle Task Hook)

Örnek 4.4'te oluşturulan görevler zamanlarının çoğunu Engellenmiş durumunda (Blocked state) geçirirler. Bu durumdayken çalışamazlar, bu nedenle çizgeleyici (scheduler) tarafından seçilemezler.

Çalışıyor (Running) durumuna girebilecek her zaman en az bir görev olmalıdır. Bunun böyle olmasını sağlamak için çizgeleyici, `vTaskStartScheduler()` çağrıldığında otomatik olarak bir Boş (Idle) görev oluşturur. Boş görev, bir döngü içinde oturmaktan çok daha fazlasını yapmaz, bu nedenle ilk örneklerdeki görevler gibi her zaman çalışabilir durumdadır.

Boş görev, daha yüksek öncelikli bir uygulama görevinin Çalışıyor durumuna girmesini asla engellememesini sağlamak için mümkün olan en düşük önceliğe (öncelik sıfır) sahiptir. Bununla birlikte, istenirse, uygulama tasarımcılarının boş görev önceliğinde ve dolayısıyla o önceliği paylaşan görevler oluşturmasını engelleyen hiçbir şey yoktur. `FreeRTOSConfig.h` içindeki `configIDLE_SHOULD_YIELD` derleme zamanı yapılandırma sabiti, Boş görevin, kendisi de o önceliğine sahip olan uygulama görevlerine daha verimli bir şekilde tahsis edilecek işlem süresini tüketmesini (consume) önlemek için

kullanılabilir. Bölüm 4.12, Çizgeleme Algoritmaları (Scheduling Algorithms), `configIDLE_SHOULD_YIELD` sabitini açıklamaktadır.

En düşük öncelikte çalışmak, daha yüksek öncelikli bir görev Hazır durumuna girer girmez Boş görevin Çalışıyor durumundan çıkarılmasını sağlar. Bu, Şekil 4.9'da tn zamanında, Görev 2'nin Engellenmiş durumundan çıktığı anda yürütülmesine izin vermek için Boş görevin hemen değiştirildiği (swapped out) yerde görülebilir. Görev 2'nin boş görevi engellediği (preempted) söylenir. Engelleme (Preemption) otomatik olarak ve engellenen görevin haberi olmadan gerçekleşir.

Not: Bir görev kendini silmek için `vTaskDelete()` API işlevini kullanırsa, Boş görevin işlem süresinden mahrum kalmaması (starved of processing time) çok önemlidir. Bunun nedeni, kendi kendini silen görevler tarafından kullanılan çekirdek kaynaklarını temizlemekten Boş görevin sorumlu olmasıdır.

4.8.1 Boş Görev Kancası İşlevleri (Idle Task Hook Functions)

Boş görev döngüsünün her yinelemesinde boş görev tarafından otomatik olarak çağrılan bir işlev olan boş kanca (veya boş geri çağırma - idle callback) işlevinin kullanılması yoluyla uygulamaya özgü işlevselliği doğrudan boş göreve eklemek mümkündür.

Boş görev kancasının (Idle task hook) yaygın kullanım alanları şunlardır:

- Düşük öncelikli, arka plan (background) veya sürekli işleme (continuous processing) işlevlerini, bu amaçla uygulama görevleri oluşturmanın getirdiği RAM yükü (overhead) olmadan yürütmek.
- Boşta kalan işlem kapasitesinin miktarını ölçmek. (Boş görev yalnızca daha yüksek öncelikli tüm uygulama görevlerinin yapacak işi kalmadığında çalışır; bu nedenle boş göreve tahsis edilen işlem süresinin miktarını ölçmek boş işlem süresinin net bir göstergesini sağlar).
- İşlemciyi düşük güç moduna yerleştirmek, gerçekleştirilecek hiçbir uygulama işlemi olmadığında (elde edilebilir güç tasarrufu tick-less boşta modunun sağladığı güç tasarrufundan daha az olsa da) kolay ve otomatik bir güç tasarrufu yöntemi sağlar.

4.8.2 Boş Görev Kancası İşlevlerinin Uygulanmasındaki Sınırlamalar (Limitations)

Boş görev kancası (Idle task hook) işlevleri aşağıdaki kurallara uymalıdır:

- Boş görev kanca işlevi asla kendisini engellemeye (block) veya askıya almaya (suspend) çalışmamalıdır.
  Not: Boş görevi herhangi bir şekilde engellemek, hiçbir görevin Çalışıyor durumuna geçemediği bir senaryoya neden olabilir.
- Bir uygulama görevi kendi kendini silmek için `vTaskDelete()` API işlevini kullanırsa, Boş görev kancası (Idle task hook) her zaman makul bir süre içinde çağıranına (caller) dönmelidir. Bunun nedeni, kendi kendini silen görevlere tahsis edilen çekirdek kaynaklarının temizlenmesinden Boş görevin (Idle task)

sorumlu olmasıdır. Boş görev kalıcı olarak Boş kanca (Idle hook) işlevinde kalırsa, bu temizleme (clean-up) işlemi gerçekleştirilemez.

- Boş görev kancası işlevleri Liste 4.18'de gösterilen isme ve prototipe sahip olmalıdır.

```
void vApplicationIdleHook( void );
```

Liste 4.18 Boş görev kanca (idle task hook) işlevi adı ve prototipi

Örnek 4.7 *Bir boş görev kanca işlevi (idle task hook function) tanımlama

Örnek 4.4'te engelleyen (blocking) `vTaskDelay()` API çağrılarının kullanılması, her iki uygulama görevinin de Engellenmiş durumunda olması nedeniyle Boş (Idle) görevin yürütüldüğü çok fazla boş zaman (idle time) yaratmıştır. Örnek 4.7, kaynağı Liste 4.19'da gösterilen bir Boş kanca işlevinin eklenmesiyle bu boş zamanı kullanır.

```
/* Kanca işlevi tarafından artırılacak (incremented) bir değişken bildirin. */

volatile unsigned long ulIdleCycleCount = 0UL;



/*

 * Boş kanca (Idle hook) işlevleri vApplicationIdleHook() olarak adlandırılmalı,

 * hiçbir parametre almamalı ve void döndürmelidir.

 */

void vApplicationIdleHook( void )

{

    /* Bu kanca işlevi, bir sayacı artırmaktan başka bir şey yapmaz. */

    ulIdleCycleCount++;

}
```

Liste 4.19 Çok basit bir Boş (Idle) kanca işlevi

Boş kanca işlevinin çağrılması için `FreeRTOSConfig.h` dosyasında `configUSE_IDLE_HOOK` 1 olarak ayarlanmalıdır.

Oluşturulan görevleri uygulayan işlev, Liste 4.20'de gösterildiği gibi `ulIdleCycleCount` değerini yazdıracak şekilde biraz değiştirilmiştir.

```
void vTaskFunction( void * pvParameters )
```

```
{
    char * pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* Yazdırılacak dize parametre aracılığıyla aktarılır. */
    pcTaskName = ( char * ) pvParameters;

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde uygulanır. */
    for( ;; )
    {
        /*
         * Bu görevin adını VE ulIdleCycleCount'un kaç kez
         * artırıldığını yazdırın.
         */
        vPrintLineAndNumber( pcTaskName, ulIdleCycleCount );

        /* 250 milisaniyelik bir süre gecikme. */
        vTaskDelay( xDelay250ms );
    }
}
```

Liste 4.20 Örnek görevin kaynak kodu artık `ulIdleCycleCount` değerini yazdırmaktadır

Şekil 4.13 Örnek 4.7 tarafından üretilen çıktıyı göstermektedir. Boş görev kancası işlevinin, uygulama görevlerinin her yinelemesi arasında yaklaşık 4 milyon kez yürütüldüğü görülebilir (yineleme sayısı donanım hızına bağlıdır).

```
C:\Temp>rtosdemo
Task 2 is running
ulIdleCycleCount = 0
Task 1 is running
ulIdleCycleCount = 0
Task 2 is running
ulIdleCycleCount = 3869504
Task 1 is running
ulIdleCycleCount = 3869504
Task 2 is running
ulIdleCycleCount = 8564623
Task 1 is running
ulIdleCycleCount = 8564623
Task 2 is running
ulIdleCycleCount = 13181489
Task 1 is running
ulIdleCycleCount = 13181489
Task 2 is running
ulIdleCycleCount = 17838406
Task 1 is running
ulIdleCycleCount = 17838406
Task 2 is running
```

Şekil 4.13 Örnek 4.7 yürütüldüğünde üretilen çıktı

4.9 Bir Görevin Önceliğini Değiştirme

4.9.1 vTaskPrioritySet() API İşlevi

`vTaskPrioritySet()` API işlevi, çizgeleyici başlatıldıktan sonra bir görevin önceliğini değiştirir. `vTaskPrioritySet()` API işlevi yalnızca `FreeRTOSConfig.h` içinde `INCLUDE_vTaskPrioritySet` 1 olarak ayarlandığında kullanılabilir.

```
void vTaskPrioritySet( TaskHandle_t xTask,

                       UBaseType_t uxNewPriority );
```

Liste 4.21 `vTaskPrioritySet()` API işlev prototipi

`vTaskPrioritySet()` parametreleri:

- **pxTask:** Önceliği değiştirilen görevin (özne görev - subject task) tanıtıcısı (handle). Görevlere yönelik tanıtıcıları (handles) elde etme hakkında bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine veya `xTaskCreateStatic()` API işlevinin dönüş değerine bakın. Bir görev, geçerli bir görev tanıtıcısı (handle) yerine `NULL` ileterek kendi önceliğini değiştirebilir.
- **uxNewPriority:** Özne görevin (subject task) ayarlanacağı öncelik. Bu değer, `configMAX_PRIORITIES`'in `FreeRTOSConfig.h` başlık dosyasında ayarlanan bir

derleme zamanı sabiti olduğu (`configMAX_PRIORITIES` - 1) maksimum kullanılabilir öncelik ile otomatik olarak sınırlandırılır (capped).

4.9.2 uxTaskPriorityGet() API İşlevi

`uxTaskPriorityGet()` API işlevi, bir görevin önceliğini döndürür. `uxTaskPriorityGet()` API işlevi yalnızca `FreeRTOSConfig.h` içinde `INCLUDE_uxTaskPriorityGet` 1 olarak ayarlandığında kullanılabilir.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t xTask );
```

Liste 4.22 `uxTaskPriorityGet()` API işlev prototipi

`uxTaskPriorityGet()` parametreleri ve dönüş değeri:

- **pxTask:** Önceliği sorgulanan görevin (özne görev) tanıtıcısı (handle). Görevlere yönelik tanıtıcıların (handles) nasıl elde edileceğine ilişkin bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine bakın. Bir görev, geçerli bir görev tanıtıcısı yerine `NULL` ileterek kendi önceliğini sorgulayabilir.
- **Dönüş değeri:** Sorgulanan göreve o anda atanmış olan öncelik.

Örnek 4.8 Görev önceliklerini değiştirme

Çizgeleyici her zaman Çalışıyor (Running) durumuna girecek görev olarak en yüksek Hazır (Ready) durumundaki görevi seçer. Örnek 4.8, iki görevin önceliğini birbirlerine göre değiştirmek için `vTaskPrioritySet()` API işlevini kullanarak bunu gösterir.

Örnek 4.8, iki farklı öncelikte iki görev oluşturur. İki görev de Engellenmiş durumuna girmelerine neden olabilecek herhangi bir API işlev çağrısı yapmaz, bu nedenle her ikisi de her zaman Hazır durumunda veya Çalışıyor durumundadır. Bu nedenle, nispi önceliği en yüksek olan görev, her zaman çizgeleyici tarafından Çalışıyor durumunda olacak şekilde seçilecektir.

Örnek 4.8 aşağıdaki gibi davranır:

1. Görev 1 (Liste 4.23) en yüksek öncelik ile oluşturulur, böylece ilk olarak çalışması garanti edilir. Görev 1, Görev 2'nin (Liste 4.24) önceliğini kendi önceliğinin üzerine çıkarmadan önce birkaç dize yazdırır.
2. Görev 2, en yüksek nispi önceliğe sahip olur olmaz çalışmaya başlar (Çalışıyor durumuna girer). Herhangi bir zamanda yalnızca bir görev Çalışıyor durumunda olabilir, bu nedenle Görev 2 Çalışıyor durumundayken, Görev 1 Hazır durumundadır.
3. Görev 2 kendi önceliğini tekrar Görev 1'in altına düşürmeden önce bir mesaj yazdırır.
4. Görev 2 önceliğini tekrar düşürdüğünde, Görev 1 bir kez daha en yüksek öncelikli görev olur, bu nedenle Görev 1 yeniden Çalışıyor durumuna girer ve Görev 2'yi tekrar Hazır durumuna zorlar.

```
void vTask1( void * pvParameters )

{

    UBaseType_t uxPriority;


    /*

     * Bu görev daha yüksek öncelik ile oluşturulduğundan her zaman Görev 2'den

     * önce çalışacaktır. Ne Görev 1 ne de Görev 2 asla engellenmez,

     * bu nedenle her ikisi de her zaman Çalışıyor veya Hazır durumunda olacaktır.

     */


    /*

     * Bu görevin çalıştığı önceliği sorgulayın - NULL aktarmak

     * "çağıran görevin önceliğini döndür" anlamına gelir.

     */

    uxPriority = uxTaskPriorityGet( NULL );


    for( ;; )

    {

        /* Bu görevin adını yazdırın. */

        vPrintLine( "Task 1 is running" );


        /*

         * Görev 2'nin önceliğini Görev 1'in üzerine çıkarmak Görev 2'nin derhal

         * çalışmaya başlamasına neden olacaktır (çünkü o zaman Görev 2
oluşturulan

         * iki görevden daha yüksek önceliğe sahip olacaktır). vTaskPrioritySet()

         * çağrısında görev 2'nin tanıtıcısının (xTask2Handle) kullanıldığına

         * dikkat edin. Liste 4.25 tanıtıcının nasıl elde edildiğini
göstermektedir.

         */
```

```
        vPrintLine( "About to raise the Task 2 priority" );

        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );


        /*
         * Görev 1 sadece Görev 2'den daha yüksek bir önceliğe sahip olduğunda
         * çalışacaktır. Bu nedenle, bu görevin bu noktaya ulaşması için Görev
2'nin
         * önceden yürütülmüş olması ve önceliğini bu görevin önceliğinin altına
         * düşürmüş olması gerekir.
         */
    }
}
```

Liste 4.23 Örnek 4.8'deki Görev 1'in uygulaması

```
void vTask2( void * pvParameters )
{
    UBaseType_t uxPriority;


    /*
     * Görev 1 daha yüksek öncelik ile oluşturulduğundan, Görev 1 her zaman
     * bu görevden önce çalışacaktır. Ne Görev 1 ne de Görev 2 asla engellenmez,
     * bu nedenle her zaman Çalışıyor veya Hazır durumunda olacaklardır.
     *
     * Bu görevin çalıştığı önceliği sorgulayın - NULL aktarmak
     * "çağıran görevin önceliğini döndür" anlamına gelir.
     */
    uxPriority = uxTaskPriorityGet( NULL );


    for( ;; )
    {
```

```
        /*
         * Bu görevin bu noktaya ulaşabilmesi için Görev 1'in halihazırda
         * çalışmış ve bu görevin önceliğini kendininkinden daha yükseğe
         * ayarlamış olması gerekir.
         */

        /* Bu görevin adını yazdırın. */
        vPrintLine( "Task 2 is running" );

        /*
         * Bu görevin önceliğini orijinal değerine geri getirin (düşürün).
         * Görev tanıtıcısı olarak NULL iletmek "çağıran görevin önceliğini
değiştir"
         * anlamına gelir. Önceliğin Görev 1'in altına ayarlanması, Görev 1'in
         * bu görevi engelleyerek (preempting) hemen tekrar çalışmaya başlamasına
         * neden olacaktır.
         */
        vPrintLine( "About to lower the Task 2 priority" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Liste 4.24 Örnek 4.8'deki Görev 2'nin uygulaması

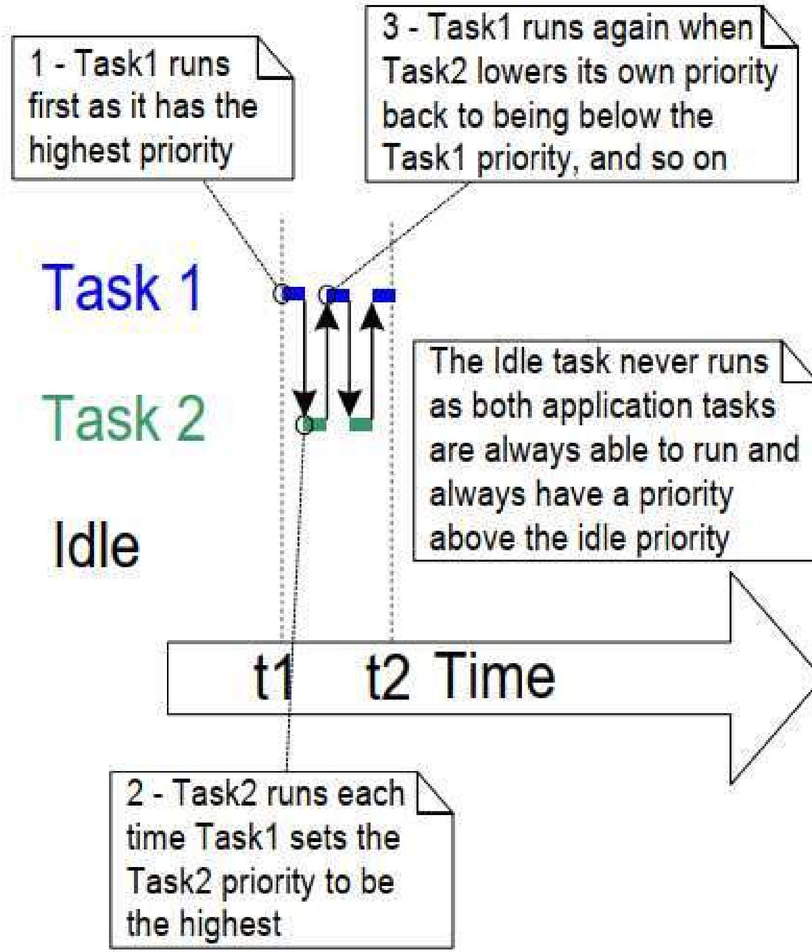
Her görev, geçerli bir görev tanıtıcısı (handle) yerine `NULL` kullanarak kendi önceliğini hem sorgulayabilir hem de ayarlayabilir. Bir görev tanıtıcısı (handle) yalnızca bir görev Görev 1'in Görev 2'nin önceliğini değiştirdiği zamanlarda olduğu gibi kendisinden başka bir göreve referans vermek istediğinde gereklidir. Görev 1'in bunu yapmasına izin vermek için, Liste 4.25'teki açıklamalarda vurgulandığı gibi Görev 2 oluşturulduğunda Görev 2'nin tanıtıcısı (handle) alınır ve kaydedilir.

```
/* Görev 2'nin tanıtıcısını (handle) tutmak için kullanılacak bir değişken
bildirin. */

TaskHandle_t xTask2Handle = NULL;
```

```
int main( void )
{
    /*
     * İlk görevi 2 önceliğinde (priority 2) oluşturun. Görev parametresi
     * kullanılmıyor ve NULL olarak ayarlanmış. Görev tanıtıcısı da
     * kullanılmıyor, dolayısıyla o da NULL olarak ayarlanmış.
     */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

    /*
     * İkinci görevi 1 önceliğinde oluşturun - bu Görev 1'e verilen
     * öncelikten daha düşüktür. Yine görev parametresi kullanılmıyor, bu nedenle
NULL olarak
     * ayarlandı, AMA bu kez görev tanıtıcısı (handle) gerekli, bu nedenle
xTask2Handle'ın
     * adresi son parametrede iletiliyor.
     */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

    /* Çizgeleyiciyi başlatın, böylece görevler yürütülmeye başlar. */
    vTaskStartScheduler();

    for( ;; )
    {
    }
}
```

Liste 4.25 Örnek 4.8 için `main()` uygulaması

Şekil 4.14 Örnek 4.8'deki görevlerin hangi sırayla yürütüldüğünü, elde edilen çıktı ise Şekil 4.15'te gösterilmektedir.



Şekil 4.14 Örnek 4.8 çalıştırılırken görevin yürütülme sırası

```
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
```

Şekil 4.15 Örnek 4.8 yürütüldüğünde üretilen çıktı

4.10 Bir Görevi Silme

4.10.1 vTaskDelete() API İşlevi

`vTaskDelete()` API işlevi bir görevi siler. `vTaskDelete()` API işlevi yalnızca `FreeRTOSConfig.h` içinde `INCLUDE_vTaskDelete` 1 olarak ayarlandığında kullanılabilir.

Çalışma zamanında sürekli olarak görev oluşturmak ve silmek iyi bir uygulama değildir, bu nedenle bu işleve ihtiyaç duyduğunuzu fark ederseniz, görevleri yeniden kullanmak gibi diğer tasarım seçeneklerini göz önünde bulundurun.

Silinen görevler artık mevcut değildir ve tekrar Çalışıyor (Running) durumuna giremezler.

Dinamik bellek tahsisi kullanılarak oluşturulan bir görev daha sonra kendi kendini silerse, silinen görevin veri yapısı ve yığını (stack) gibi kullanım için ayrılan belleği serbest bırakmaktan Boş (Idle) görev sorumludur. Bu nedenle uygulamaların bu durumdayken Boş görevi tüm işlem süresinden tamamen mahrum bırakmaması (starve etmemesi) önemlidir.

Not: Sadece çekirdeğin kendisi tarafından bir göreve tahsis edilen bellek görev silindiğinde otomatik olarak serbest bırakılır. Görevin yürütülmesi sırasında tahsis edilen herhangi bir bellek veya diğer kaynak, artık ihtiyaç duyulmuyorsa açıkça serbest bırakılmalıdır.

```
void vTaskDelete( TaskHandle_t xTaskToDelete );
```

Liste 4.26 `vTaskDelete()` API işlev prototipi

`vTaskDelete()` parametreleri:

- **p`xTaskToDelete`:** Silinecek görevin (özne görev) tanıtıcısı (handle). Görevlere yönelik tanıtıcıların (handles) elde edilmesine ilişkin bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine ve `xTaskCreateStatic()` API işlevinin dönüş değerine bakın. Bir görev, geçerli bir görev tanıtıcısı yerine `NULL` ileterek kendi kendini silebilir.

Örnek 4.9 Görevleri silme

Bu, aşağıdaki gibi davranan çok basit bir örnektir.

1. Görev 1, öncelik 1 ile `main()` tarafından oluşturulur. Çalıştığında, öncelik 2'de Görev 2'yi oluşturur. Görev 2 artık en yüksek öncelikli görevdir, bu nedenle derhal yürütülmeye başlar. Liste 4.27 `main()` için kaynak kodunu gösterir. Liste 4.28 Görev 1'in kaynak kodunu gösterir.
2. Görev 2 kendini silmekten başka bir şey yapmaz. `vTaskDelete()`'e `NULL` parametresini geçirerek kendi kendini silebilir, ancak bunun yerine, gösterim

(demonstration) amacıyla, kendi görev tanıtıcısını (handle) kullanır. Liste 4.29 Görev 2'nin kaynak kodunu göstermektedir.

3. Görev 2 silindiğinde, Görev 1 tekrar en yüksek öncelikli görev olur, bu nedenle yürütülmeye devam eder - bu noktada kısa bir süre için engellenmek üzere `vTaskDelay()` işlevini çağırır.
4. Görev 1 Engellenmiş durumundayken Boş (Idle) görev çalışır ve artık silinmiş olan Görev 2'ye tahsis edilmiş belleği serbest bırakır.
5. Görev 1 Engellenmiş durumundan çıktığında tekrar en yüksek öncelikli Hazır durumdaki görev olur ve Boş görevi engeller (preempts). Çalışıyor durumuna girdiğinde tekrar Görev 2'yi oluşturur ve bu şekilde devam eder.

```
int main( void )
{
    /* İlk görevi öncelik 1 ile oluşturun. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* Çizgeleyiciyi başlatın, böylece görev yürütülmeye başlar. */
    vTaskStartScheduler();

    /* Çizgeleyici başlatıldığı için main() asla buraya ulaşmamalıdır. */
    for( ;; )
    {
    }
}
```

Liste 4.27 Örnek 4.9 için `main()` işlevinin uygulanması

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void * pvParameters )
{
    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )
```

```
    {
        /* Bu görevin adını yazdırın. */
        vPrintLine( "Task 1 is running" );

        /*
         * Görev 2'yi daha yüksek bir öncelikte oluşturun.
         * xTaskCreate'in elde edilen görev tanıtıcısını (handle) o değişkene
         * yazması için pxCreatedTask parametresi olarak xTask2Handle'ın
         * adresini iletin.
         */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /*
         * Görev 2 daha yüksek önceliğe sahiptir/sahipti. Görev 1'in buraya
         * ulaşması için Görev 2'nin yürütülmüş ve kendi kendini silmiş
         * olması gerekir.
         */
        vTaskDelay( xDelay100ms );
    }
}
```

Liste 4.28 Örnek 4.9 için Görev 1'in uygulanması

```
void vTask2( void * pvParameters )
{
    /*
     * Görev 2 başlar başlamaz kendi kendini siler.
     * Bunu yapmak için parametre olarak NULL kullanarak vTaskDelete()
     * işlevini çağırabilirdi. Ancak gösterim (demonstration) amacıyla
     * vTaskDelete() işlevini kendi görev tanıtıcısı (handle) ile çağırır.
```

```
     */

    vPrintLine( "Task 2 is running and about to delete itself" );

    vTaskDelete( xTask2Handle );

}
```

```
C:\Temp>rtosdemo
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
```

Liste 4.29 Örnek 4.9 için Görev 2'nin uygulanması

4.11 İş Parçacığı Yerel Depolaması (Thread Local Storage) ve Yeniden Giriş (Reentrancy)

Bölüm 4.2, C çalışma zamanı (C Runtime) ortamının görev yığınlarını (task stacks) nasıl yönettiğini açıklamıştır. Bu, her görevin *auto* veya yığın-tahsisli (stack-allocated) değişkenlerin kendi benzersiz kopyasına sahip olduğu anlamına gelir. C çalışma zamanı, statik (static) değişkenler ile aynı yöntemi kullanarak bir göreve özel değişken oluşturulmasını desteklemez. Bazı karmaşık uygulamalar ve derleyiciye özgü çalışma zamanı C kütüphanelerinin (library) tüm sistemde değil, aynı görevin ardışık çağrıları boyunca kalıcı olan özel değişkenlere (private variables) erişmesi gerekir.

Bunu ele almak için FreeRTOS, görevlerin bu tür özel değerleri depolamasına ve okumasına olanak tanıyan İş Parçacığı Yerel Depolamasına (Thread Local Storage - TLS) sahiptir. Uygulama ve derleyici bu amaç için tahsis edilen alanı üç farklı şekilde kullanabilir:

1. C Çalışma Zamanı (C Runtime) TLS Uygulamaları
2. Özel (Custom) C Çalışma Zamanı TLS
3. Uygulama TLS

4.11.1 C Çalışma Zamanı (C Runtime) TLS Uygulamaları

Uygulamanız standart bir derleyici sağlayan C çalışma zamanı kütüphanesini kullanıyorsa ve kütüphane TLS'yi destekliyorsa, destekleyici TLS yapıları (structures) `configUSE_C_RUNTIME_WORKAROUND`, `FreeRTOSConfig.h`'de 1 olarak ayarlanarak kullanılabilir. Bu, standart kütüphanenin, uygulamanın FreeRTOS aracılığıyla haberdar olacağı özel TLS bloklarını (TLS blocks) tahsis etmesini ve yönetmesini sağlayacaktır.

4.11.2 Özel (Custom) C Çalışma Zamanı TLS

Eğer C Çalışma Zamanı kütüphanesi (C Runtime library) TLS desteklemiyorsa, uygulama yazarının `FreeRTOSConfig.h`'de `configUSE_CUSTOM_C_RUNTIME_TLS` öğesini 1 olarak ayarlayarak ve aşağıdaki işlevleri (functions) sağlayarak kendi TLS yöntemini oluşturması mümkündür:

- `vPortConfigCustomCRuntimeTLS()` - Görev tahsis edilirken gereken herhangi bir yapılandırmayı (configuration) ayarlar.
- `vPortCleanUpCustomCRuntimeTLS()` - Görev tahsisi kaldırılırken/silinirken gereken herhangi bir temizliği (cleanup) yapar.

4.11.3 Uygulama TLS (Application TLS)

Tüm RTOS API işlevlerinde olduğu gibi bir uygulamanın İş Parçacığı Yerel Depolamayı (Thread Local Storage - TLS) kullanmak için uygun şekilde yapılandırılması gerekir. Bir TLS dizesine (array) erişim sağlamak için `FreeRTOSConfig.h` içindeki `configNUM_THREAD_LOCAL_STORAGE_POINTERS` yapılandırma sabiti dizede saklanabilecek en büyük işaretçi sayısına ayarlanmalıdır.

Örneğin, Görev A ve Görev B `vTaskFunction()` adında ortak bir işlev kullanıyorsa, işlev hangi görevin yürütüldüğüne bağlı olarak her görevin tahsis edilen İş Parçacığı Yerel Depolama (TLS) işaretçisinden belirli verileri okuyabilir. `vTaskSetThreadLocalStoragePointer()` ve `pvTaskGetThreadLocalStoragePointer()` işlevleri bir görevin İş Parçacığı Yerel Depolamasını sırasıyla yazmak ve okumak için kullanılır.

```
void vTaskSetThreadLocalStoragePointer( TaskHandle_t xTaskToSet,

                                        BaseType_t xIndex,

                                        void * pvValue );
```

Liste 4.30 `vTaskSetThreadLocalStoragePointer()` API işlev prototipi

`vTaskSetThreadLocalStoragePointer()` parametreleri:

- `xTaskToSet`**:** TLS alanına değer atayan görevin tanıtıcısı (handle). Eğer `NULL` ise, o zaman çağıran görevin TLS alanına bir değer atar.
- **xIndex:** Saklanacak değerin TLS dizinidir.
- **pvValue:** Değer tahsis edilmiş bir işaretçiye yazılacaktır.

```
void * pvTaskGetThreadLocalStoragePointer( TaskHandle_t xTaskToQuery,
                                           BaseType_t xIndex );
```

Liste 4.31 `pvTaskGetThreadLocalStoragePointer()` API işlev prototipi

`pvTaskGetThreadLocalStoragePointer()` **parametreleri:**

- `xTaskToQuery`**:** TLS alanını okuyan görevin tanıtıcısı (handle). Eğer `NULL` ise, o zaman çağıran görevin TLS alanını okur.
- **xIndex:** Okunacak değerin TLS dizinidir (index).

4.12 Çizgeleme Algoritmaları (Scheduling Algorithms)

4.12.1 Görev Durumları ve Olayların Özeti

Bir görev aşağıdaki durumlardan birinde bulunabilir:

- **Çalışıyor (Running):** Kod yürütülüyor demektir.
- **Hazır (Ready):** Görev çalışmaya hazırdır ve yürütülmeyi bekler.
- **Engellenmiş (Blocked):** Görev bir olay beklemektedir (örneğin zaman aşımı veya kuyruk bekleme).
- **Askıya Alınmış (Suspended):** Görev tamamen durdurulmuştur ve özel olarak devam ettirilmesi (resume) gerekir.

Görev durumları, API işlevi çağrıları, donanım kesmeleri (interrupts) ve zaman geçişleri nedeniyle değişir.

4.12.2 Çizgeleme Algoritmasını Seçme

FreeRTOS'un çalıştıracağı algoritmayı `FreeRTOSConfig.h`'deki yapılandırma ayarları belirler. `configUSE_PREEMPTION` ve `configUSE_TIME_SLICING` bu davranışı belirler.

- `configUSE_PREEMPTION` **= 1:** Öncelikli ve Ön Alımlı (Preemptive) çizgeleme.
- `configUSE_TIME_SLICING` **= 1:** Zaman Dilimlemeli (Time Slicing) çizgeleme.

4.12.3 Zaman Dilimleme ile Öncelikli Ön Alımlı Çizgeleme (Prioritized Preemptive Scheduling with Time Slicing)

Aşağıdaki tabloda gösterilen yapılandırma, FreeRTOS çizgeleyicisini (scheduler) 'Zaman Dilimlemeli Sabit Öncelikli Önleyici Çizgeleme' (*Fixed Priority Preemptive Scheduling with Time Slicing*) adlı bir çizgeleme algoritmasını kullanacak şekilde ayarlar; bu, çoğu küçük RTOS uygulaması tarafından kullanılan çizgeleme algoritmasıdır ve bu kitapta şimdiye kadar sunulan tüm örneklerde kullanılan algoritmadır. Sonraki tablo, algoritmanın adında kullanılan terminolojinin bir açıklamasını sunmaktadır.

Çizgeleme politikasını tanımlamak için kullanılan terimlerin açıklaması:

• Sabit Öncelik (Fixed Priority)

'Sabit Öncelik' olarak tanımlanan çizgeleme algoritmaları, çizgelenen görevlere atanan önceliği değiştirmez, ancak görevlerin kendi önceliklerini veya diğer görevlerin önceliklerini değiştirmelerini de engellemez.

• Önleyici (Preemptive)

Önleyici çizgeleme algoritmaları, *Çalışıyor* (Running) durumundaki görevden daha yüksek önceliğe sahip bir görev *Hazır* (Ready) durumuna girerse, *Çalışıyor* durumundaki görevi anında 'önler' (preempt). Önlenmiş olmak, gönüllü olarak teslim olmadan (yielding) veya engellenmeden (blocking), *Çalışıyor* durumundan çıkarılıp *Hazır* durumuna istem dışı taşınmak demektir; böylece farklı bir görevin *Çalışıyor* durumuna girmesine izin verilir. Görev önleme, yalnızca RTOS tick kesmesinde değil, herhangi bir zamanda gerçekleşebilir.

• Zaman Dilimleme (Time Slicing)

Zaman dilimleme, görevler gönüllü olarak teslim olmasa (yield) veya *Engellenmiş* (Blocked) durumuna girmese bile, eşit önceliğe sahip görevler arasında işlem süresini paylaşmak için kullanılır. *Zaman Dilimleme* kullanıldığı açıklanan çizgeleme algoritmaları, eğer *Çalışıyor* durumundaki görevle aynı önceliğe sahip başka *Hazır* durumundaki görevler varsa, her zaman diliminin sonunda *Çalışıyor* durumuna girecek yeni bir görev seçer. Bir zaman dilimi, iki RTOS tick kesmesi arasındaki süreye eşittir.

• Sabit Öncelik (Fixed Priority)

'Sabit Öncelik' olarak tanımlanan çizgeleme algoritmaları, çizgelenen görevlere atanan önceliği değiştirmez, ancak görevlerin kendi önceliklerini veya diğer görevlerin önceliklerini değiştirmelerini de engellemez.

• Önleyici (Preemptive)

Önleyici çizgeleme algoritmaları, *Çalışıyor* (Running) durumundaki görevden daha yüksek önceliğe sahip bir görev *Hazır* (Ready) durumuna girerse, *Çalışıyor* durumundaki görevi anında 'önler' (preempt). Önlenmiş olmak,

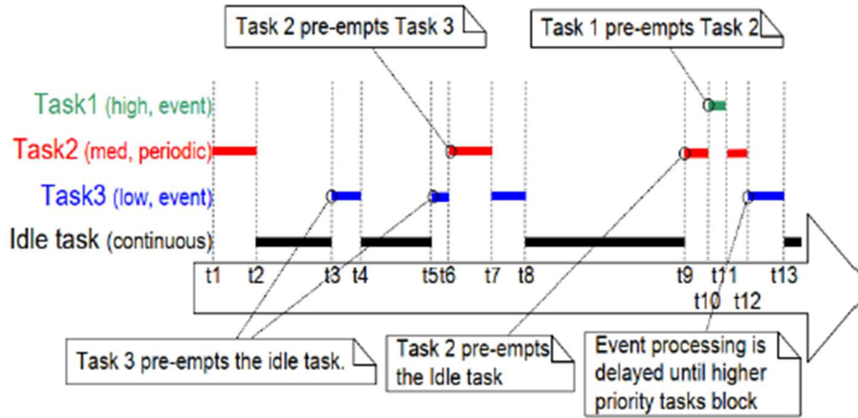
gönüllü olarak teslim olmadan (yielding) veya engellenmeden (blocking), *Çalışıyor* durumundan çıkarılıp *Hazır* durumuna istem dışı taşınmak demektir; böylece farklı bir görevin *Çalışıyor* durumuna girmesine izin verilir. Görev önleme, yalnızca RTOS tick kesmesinde değil, herhangi bir zamanda gerçekleşebilir.

• Zaman Dilimleme (Time Slicing)

Zaman dilimleme, görevler gönüllü olarak teslim olmasa (yield) veya *Engellenmiş* (Blocked) durumuna girmese bile, eşit önceliğe sahip görevler arasında işlem süresini paylaşmak için kullanılır. *Zaman Dilimleme* kullanıldığı açıklanan çizgeleme algoritmaları, eğer *Çalışıyor* durumundaki görevle aynı önceliğe sahip başka *Hazır* durumundaki görevler varsa, her zaman diliminin sonunda *Çalışıyor* durumuna girecek yeni bir görev seçer. Bir zaman dilimi, iki RTOS tick kesmesi arasındaki süreye eşittir.

Şekil 4.18 ve Şekil 4.19, zaman dilimlemeli sabit öncelikli önleyici çizgeleme algoritması kullanıldığında görevlerin nasıl çizgelendiğini göstermektedir. Şekil 4.18, bir uygulamadaki tüm görevlerin benzersiz bir önceliğe sahip olduğu durumda görevlerin *Çalışıyor* (Running) durumuna girmek üzere seçilme sırasını göstermektedir. Şekil 4.19, bir uygulamadaki iki görevin aynı önceliği paylaştığı durumda görevlerin *Çalışıyor* (Running) durumuna girmek üzere seçilme sırasını göstermektedir.

tasks are selected to enter the *Running* state when two tasks in an application share a priority.



Şekil 4.18 Her görevin benzersiz bir önceliğe atandığı varsayımsal bir uygulamada görev önceliklendirme ve önlemeyi (preemption) vurgulayan yürütme deseni

Şekil 4.18'e Atıfta Bulunarak:

• Boş Görev (Idle Task)

Boş görev en düşük öncelikte çalışır, bu nedenle daha yüksek öncelikli bir görev *Hazır* (Ready) durumuna her girdiğinde önlenir (preempted) — örneğin, t3, t5 ve t9 zamanlarında.

• **Görev 3 (Task 3)**

Görev 3, nispeten düşük önceliğe sahip ancak Boş önceliğinin üzerinde yürütülen olay güdümlü (event-driven) bir görevdir. Zamanının çoğunu *Engellenmiş* (Blocked) durumunda ilgilendiği olayı bekleyerek geçirir, olay her gerçekleştiğinde *Engellenmiş* durumundan *Hazır* (Ready) durumuna geçiş yapar. Tüm FreeRTOS görevler arası iletişim mekanizmaları (görev bildirimleri, kuyruklar, semaforlar, olay grupları vb.) olayları bildirmek ve görevlerin engelini kaldırmak için bu şekilde kullanılabilir.

Olaylar t3 ve t5 zamanlarında ve ayrıca t9 ile t12 arasında bir yerde gerçekleşir. t3 ve t5 zamanlarında gerçekleşen olaylar hemen işlenir, çünkü bu zamanlarda Görev 3 çalışabilen en yüksek öncelikli görevdir. t9 ile t12 arasında bir yerde gerçekleşen olay, t12'ye kadar işlenmez; çünkü o zamana kadar daha yüksek öncelikli görevler olan Görev 1 ve Görev 2 hâlâ yürütülmektedir. Ancak t12 zamanında hem Görev 1 hem de Görev 2 *Engellenmiş* durumunda olduğundan, Görev 3 en yüksek öncelikli *Hazır* durumundaki görev haline gelir.

t12'ye kadar işlenmez; çünkü o zamana kadar daha yüksek öncelikli görevler olan Görev 1 ve Görev 2 hâlâ yürütülmektedir. Ancak t12 zamanında hem Görev 1 hem de Görev 2 *Engellenmiş* durumunda olduğundan, Görev 3 en yüksek öncelikli *Hazır* durumundaki görev haline gelir.

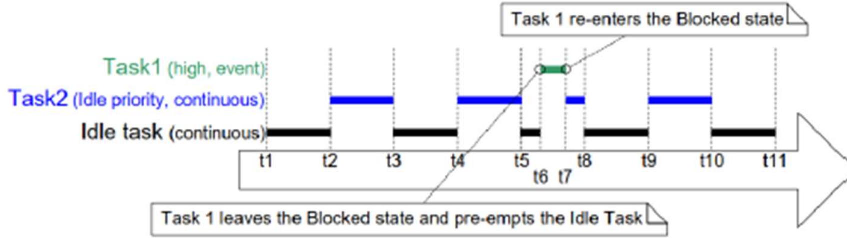
• **Görev 2 (Task 2)**

Görev 2, Görev 3'ün önceliğinin üzerinde ancak Görev 1'in önceliğinin altında yürütülen periyodik bir görevdir. Görevin periyot aralığı, Görev 2'nin t1, t6 ve t9 zamanlarında yürütülmek istediği anlamına gelir.

t6 zamanında, Görev 3 *Çalışıyor* (Running) durumundadır, ancak Görev 2 daha yüksek göreceli önceliğe sahiptir, bu nedenle Görev 3'ü önler ve hemen yürütülmeye başlar. Görev 2, işlemesini tamamlar ve t7 zamanında *Engellenmiş* (Blocked) durumuna yeniden girer, bu noktada Görev 3 işlemesini tamamlamak üzere *Çalışıyor* (Running) durumuna yeniden girebilir. Görev 3 kendisi t8 zamanında Engellenir (Blocks).

• **Görev 1 (Task 1)**

Görev 1 de olay güdümlü (event-driven) bir görevdir. Tümünün en yüksek önceliğiyle yürütülür, bu nedenle sistemdeki diğer herhangi bir görevi önleyebilir (preempt). Gösterilen tek Görev 1 olayı t10 zamanında gerçekleşir, bu noktada Görev 1, Görev 2'yi önler. Görev 2, ancak Görev 1 t11 zamanında *Engellenmiş* (Blocked) durumuna yeniden girdikten sonra işlemesini tamamlayabilir.



Şekil 4.19 İki görevin aynı öncelikte çalıştığı varsayımsal bir uygulamada görev önceliklendirme ve zaman dilimlemeyi vurgulayan yürütme deseni

Şekil 4.19'a Atıfta Bulunarak:

• Boş Görev (Idle Task) ve Görev 2 (Task 2)

Boş görev ve Görev 2 her ikisi de sürekli işleme görevleridir ve her ikisi de 0 (mümkün olan en düşük) önceliğe sahiptir. Çizgeleyici (scheduler) yalnızca çalışabilen daha yüksek öncelikli görevler olmadığında öncelik 0 görevlerine işlem süresi ayırır ve öncelik 0 görevlerine ayrılan süreyi zaman dilimleme ile paylaştırır. Yeni bir zaman dilimi her tick kesmesinde başlar ve Şekil 4.19'da t1, t2, t3, t4, t5, t8, t9, t10 ve t11 zamanlarında gerçekleşir.

Boş görev ve Görev 2 sırayla *Çalışıyor* (Running) durumuna girerler, bu da her iki görevin de aynı zaman diliminin bir bölümünde *Çalışıyor* durumunda olmasına neden olabilir — t5 ile t8 zamanları arasında olduğu gibi.

• Görev 1 (Task 1)

Görev 1'in önceliği Boş önceliğinden yüksektir. Görev 1, zamanının çoğunu *Engellenmiş* (Blocked) durumunda ilgilendiği olayı bekleyerek geçiren olay güdümlü bir görevdir, olay her gerçekleştiğinde *Engellenmiş* durumundan *Hazır* (Ready) durumuna geçiş yapar.

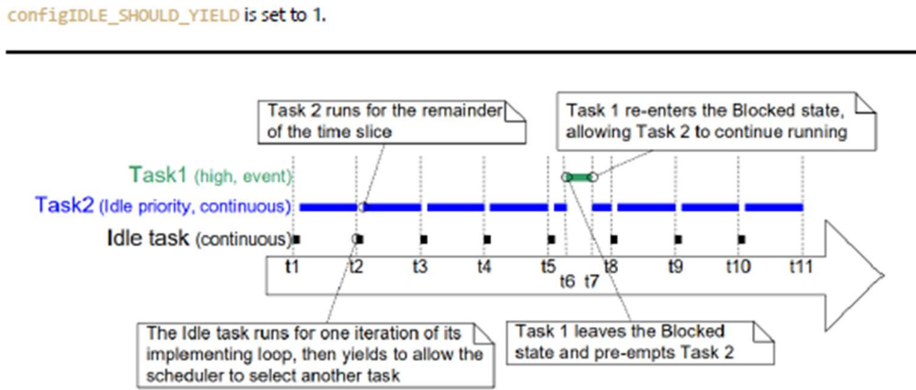
İlgilenilen olay t6 zamanında gerçekleşir. t6 zamanında Görev 1 çalışabilen en yüksek öncelikli görev haline gelir ve bu nedenle Görev 1 bir zaman diliminin ortasında Boş görevi önler. Olay işleme t7 zamanında tamamlanır, bu noktada Görev 1 *Engellenmiş* (Blocked) durumuna yeniden girer.

Şekil 4.19, Boş görevin uygulama yazarı tarafından oluşturulan bir görevle işlem süresini paylaştığını göstermektedir. Boş göreve bu kadar çok işlem süresi ayırmak, uygulama yazarı tarafından oluşturulan Boş öncelikli görevlerin yapacak işi varsa ama Boş görevin yoksa istenmeyebilir. configIDLE_SHOULD_YIELD derleme zamanı yapılandırma sabiti, Boş görevin nasıl çizgeleneceğini değiştirmek için kullanılabilir:

• Eğer configIDLE_SHOULD_YIELD 0 olarak ayarlanırsa, Boş görev daha yüksek öncelikli bir görev tarafından önlenmedikçe zaman diliminin tamamı boyunca *Çalışıyor* (Running) durumunda kalır.

• Eğer configIDLE_SHOULD_YIELD 1 olarak ayarlanırsa, *Hazır* (Ready) durumunda başka Boş öncelikli görevler varsa, Boş görev döngüsünün her yinelemesinde teslim olur (yield) (kalan zaman diliminin ne kadarı varsa gönüllü olarak bırakır).

Şekil 4.19'da gösterilen yürütme deseni, configIDLE_SHOULD_YIELD 0 olarak ayarlandığında gözlemlenecek olan desendir. Şekil 4.20'de gösterilen yürütme deseni, aynı senaryoda configIDLE_SHOULD_YIELD 1 olarak ayarlandığında gözlemlenecek olan desendir.



Şekil 4.20 Şekil 4.19'dakiyle aynı senaryo için yürütme deseni, ancak bu sefer configIDLE_SHOULD_YIELD *1 olarak ayarlanmış*

Şekil 4.20 ayrıca, configIDLE_SHOULD_YIELD 1 olarak ayarlandığında, Boş görevden sonra *Çalışıyor* (Running) durumuna girmek üzere seçilen görevin tam bir zaman dilimi boyunca yürütülmediğini, bunun yerine Boş görevin teslim olduğu zaman diliminin kalan süresi boyunca yürütüldüğünü göstermektedir.

4.12.4 Zaman Dilimlemesi Olmadan Öncelikli Ön Alımlı Çizgeleme

Zaman dilimlemesi olmadan öncelikli önleyici çizgeleme, önceki bölümde açıklanan görev seçimi ve önleme (preemption) algoritmalarının aynısını sürdürür, ancak eşit önceliğe sahip görevler arasında işlem süresini paylaşmak için zaman dilimleme kullanmaz.

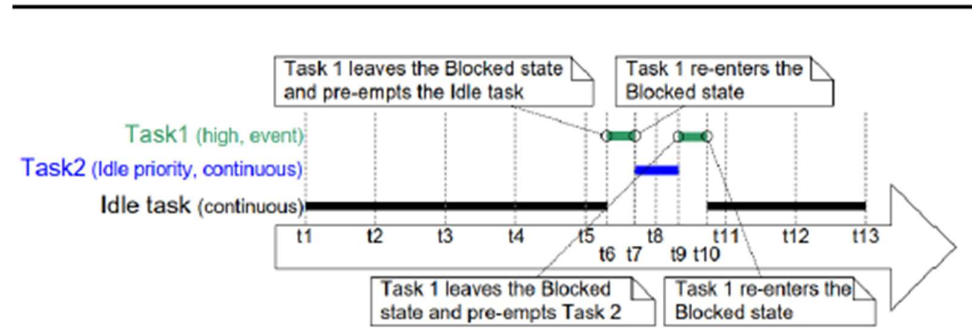
Aşağıdaki tablo, FreeRTOS çizgeleyicisini (scheduler) zaman dilimlemesi olmadan öncelikli önleyici çizgeleme kullanacak şekilde yapılandıran FreeRTOSConfig.h ayarlarını göstermektedir.

Şekil 4.19'da gösterildiği gibi, zaman dilimleme kullanılıyorsa ve çalışabilen en yüksek öncelikte birden fazla hazır durumdaki görev varsa, çizgeleyici her RTOS tick kesmesinde (bir tick kesmesi bir zaman diliminin sonunu işaretler) *Çalışıyor* (Running) durumuna girecek yeni bir görev seçer. Zaman dilimleme kullanılmıyorsa, çizgeleyici yalnızca aşağıdaki durumlarda *Çalışıyor* durumuna girecek yeni bir görev seçer:

• Daha yüksek öncelikli bir görev *Hazır* (Ready) durumuna girer.

• *Çalışıyor* (Running) durumundaki görev *Engellenmiş* (Blocked) veya *Askıya Alınmış* (Suspended) durumuna girer.

Zaman dilimleme kullanılmadığında, zaman dilimleme kullanıldığına göre daha az görev bağlam anahtarı (context switch) gerçekleşir. Bu nedenle, zaman dilimlemeyi kapatmak çizgeleyicinin işleme yükünü azaltır. Ancak zaman dilimlemeyi kapatmak, eşit önceliğe sahip görevlerin büyük ölçüde farklı miktarlarda işlem süresi almasına da neden olabilir — Şekil 4.21 tarafından gösterilen bir senaryo. Bu nedenle, çizgeleyiciyi zaman dilimleme olmadan çalıştırmak, yalnızca deneyimli kullanıcılar tarafından kullanılması gereken ileri düzey bir teknik olarak kabul edilir.



ure 4.21 *Execution pattern that demonstrates how tasks of equal priority can receive hugely different amounts of*

Şekil 4.21 Zaman dilimleme kullanılmadığında eşit önceliğe sahip görevlerin büyük ölçüde farklı miktarlarda işlem süresi alabileceğini gösteren yürütme deseni

Şekil 4.21'e Atıfta Bulunarak, configIDLE_SHOULD_YIELD 0 olarak ayarlanmış varsayılarak:

• **Tick Kesmeleri**

Tick kesmeleri t1, t2, t3, t4, t5, t8, t11, t12 ve t13 zamanlarında gerçekleşir.

• **Görev 1 (Task 1)**

Görev 1, zamanının çoğunu *Engellenmiş* (Blocked) durumunda ilgilendiği olayı bekleyerek geçiren yüksek öncelikli olay güdümlü bir görevdir. Görev 1 her olay gerçekleştiğinde *Engellenmiş* durumundan *Hazır* (Ready) durumuna geçiş yapar (ve ardından en yüksek öncelikli *Hazır* durumundaki görev olduğu için *Çalışıyor* — Running durumuna geçer). Şekil 4.21, Görev 1'in t6 ile t7 zamanları arasında ve tekrar t9 ile t10 zamanları arasında bir olayı işlediğini göstermektedir.

• **Boş Görev (Idle Task) ve Görev 2 (Task 2)**

Boş görev ve Görev 2 her ikisi de sürekli işleme görevleridir ve her ikisi de 0 (Boş — idle önceliği) önceliğe sahiptir. Sürekli işleme görevleri *Engellenmiş* (Blocked) durumuna girmez.

Zaman dilimleme kullanılmıyor, bu nedenle *Çalışıyor* (Running) durumundaki bir Boş öncelikli görev, daha yüksek öncelikli Görev 1 tarafından önlenene kadar *Çalışıyor* durumunda kalmaya devam eder.

Şekil 4.21'de Boş görev t1 zamanında çalışmaya başlar ve Görev 1 tarafından t6 zamanında önlenene kadar *Çalışıyor* (Running) durumunda kalır — bu, *Çalışıyor* durumuna girdikten sonra dörtten fazla tam tick periyodudur.

Görev 2, t7 zamanında çalışmaya başlar — bu, Görev 1'in başka bir olay beklemek üzere *Engellenmiş* (Blocked) durumuna yeniden girdiği zamandır. Görev 2, Görev 1 tarafından t9 zamanında önlenene kadar Çalışıyor durumunda kalır — bu, *Çalışıyor* durumuna girdikten sonra bir tick periyodundan az bir süredir.

t10 zamanında Boş görev *Çalışıyor* (Running) durumuna yeniden girer — Görev 2'den dörtten fazla kat daha fazla işlem süresi almış olmasına rağmen.

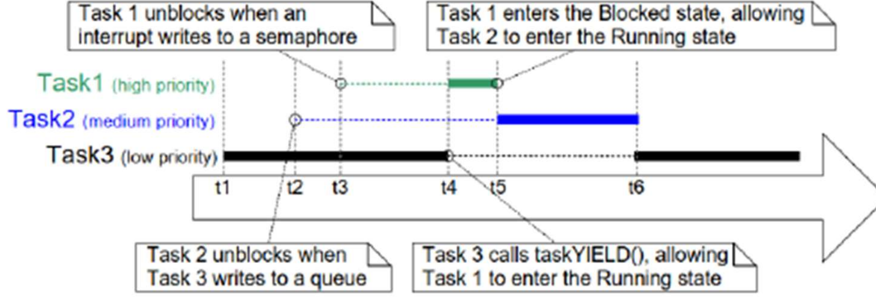
4.12.5 İşbirlikçi (Cooperative) Çizgeleme

Bu kitap önleyici (preemptive) çizgelemeye odaklanmaktadır, ancak FreeRTOS işbirlikçi (cooperative) çizgeleme de kullanabilir. Aşağıdaki tablo, FreeRTOS çizgeleyicisini işbirlikçi çizgeleme kullanacak şekilde yapılandıran FreeRTOSConfig.h ayarlarını göstermektedir.

İşbirlikçi çizgeleyici kullanılırken (ve bu nedenle uygulama tarafından sağlanan kesme hizmet rutinlerinin açıkça bağlam anahtarı talep etmediği varsayılarak), bağlam anahtarı yalnızca Çalışıyor (Running) durumundaki görev *Engellenmiş* (Blocked) durumuna girdiğinde veya *Çalışıyor* (Running) durumundaki görev taskYIELD() çağırarak açıkça teslim olduğunda (yeniden çizgeleme talep ettiğinde) gerçekleşir. Görevler asla önlenmez (preempted), bu nedenle zaman dilimleme kullanılamaz.

Şekil 4.22 işbirlikçi çizgeleyicinin davranışını göstermektedir. Şekil 4.22'deki yatay kesikli çizgiler, bir görevin *Hazır* (Ready) durumunda olduğu zamanı gösterir.

Figure 4.22 demonstrates the behavior of the cooperative scheduler. The horizontal dashed lines in Figure 4.22 show when a task is in the Ready state.



Şekil 4.22 İşbirlikçi çizgeleyicinin davranışını gösteren yürütme deseni

Şekil 4.22'ye Atıfta Bulunarak:

• **Görev 1 (Task 1)**

Görev 1 en yüksek önceliğe sahiptir. Bir semafor bekleyerek *Engellenmiş* (Blocked) durumunda başlar.

t3 zamanında, bir kesme (interrupt) semaforu verir ve Görev 1'in *Engellenmiş* durumundan çıkarak *Hazır* (Ready) durumuna girmesine neden olur (kesmelerden semafor verme Bölüm 6'da ele alınmaktadır).

t3 zamanında, Görev 1 en yüksek öncelikli *Hazır* durumundaki görevdir ve eğer önleyici (preemptive) çizgeleyici kullanılıyor olsaydı, Görev 1 Çalışıyor (Running) durumundaki görev olurdu. Ancak işbirlikçi çizgeleyici kullanıldığından, Görev 1 *Hazır* durumunda t4 zamanına kadar kalır — bu, *Çalışıyor* (Running) durumundaki görevin taskYIELD() çağırdığı zamandır.

• **Görev 2 (Task 2)**

Görev 2'nin önceliği Görev 1 ve Görev 3'ün arasındadır. *Engellenmiş* (Blocked) durumunda başlar, t2 zamanında Görev 3 tarafından kendisine gönderilen bir mesajı bekler.

t2 zamanında, Görev 2 en yüksek öncelikli *Hazır* (Ready) durumundaki görevdir ve eğer önleyici çizgeleyici kullanılıyor olsaydı, Görev 2 Çalışıyor (Running) durumundaki görev olurdu. Ancak işbirlikçi çizgeleyici kullanıldığından, Görev 2 *Hazır* durumunda *Çalışıyor* durumundaki görev *Engellenmiş* durumuna girene veya taskYIELD() çağırana kadar kalır.

Çalışıyor durumundaki görev t4 zamanında taskYIELD() çağırır, ancak bu noktada Görev 1 en yüksek öncelikli *Hazır* durumundaki görevdir, bu nedenle Görev 2 aslında Görev 1 t5 zamanında *Engellenmiş* durumuna yeniden girene kadar *Çalışıyor* durumundaki görev olmaz.

t6 zamanında, Görev 2 bir sonraki mesajı beklemek üzere *Engellenmiş* (Blocked) durumuna yeniden girer ve bu noktada Görev 3 bir kez daha en yüksek öncelikli *Hazır* durumundaki görev olur.

Çok görevli (multi-tasking) bir uygulamada, uygulama yazarı bir kaynağa aynı anda birden fazla görev tarafından erişilmemesine dikkat etmelidir, çünkü eşzamanlı erişim kaynağı bozabilir. Örnek olarak, erişilen kaynağın bir UART (seri port) olduğu aşağıdaki senaryoyu düşünün. İki görev UART'a dizeler yazar; Görev 1 "abcdefghijklmnop" yazar ve Görev 2 "123456789" yazar:

1. Görev 1 *Çalışıyor* (Running) durumundadır ve dizesini yazmaya başlar. UART'a "abcdefg" yazar, ancak başka karakter yazmadan önce *Çalışıyor* durumundan çıkar.

2. Görev 2 *Çalışıyor* (Running) durumuna girer ve *Çalışıyor* durumundan çıkmadan önce UART'a "123456789" yazar.

3. Görev 1 *Çalışıyor* (Running) durumuna yeniden girer ve dizesinin kalan karakterlerini UART'a yazar.

Bu senaryoda, UART'a gerçekte yazılan şey "abcdefg123456789hijklmnop" olur. Görev 1 tarafından yazılan dize, amaçlandığı gibi kesintisiz bir sıra halinde UART'a yazılmamıştır — bunun yerine bozulmuştur, çünkü Görev 2 tarafından UART'a yazılan dize onun içinde görünmektedir.

İşbirlikçi çizgeleyiciyi kullanmak, önleyici çizgeleyiciyi kullanmaya göre eşzamanlı erişimin neden olduğu sorunlardan kaçınmayı normalde kolaylaştırır[^7]:

[^7]: Görevler arasında kaynakları güvenle paylaşma yöntemleri bu kitapta ilerleyen bölümlerde ele alınmaktadır. FreeRTOS'un kendisi tarafından sağlanan kuyruklar ve semaforlar gibi kaynaklar, görevler arasında paylaşmak için her zaman güvenlidir.

• Önleyici çizgeleyiciyi kullandığınızda, *Çalışıyor* (Running) durumundaki görev herhangi bir zamanda önlenebilir (preempted) — bir kaynağı başka bir görevle paylaşırken tutarsız bir durumda olduğu zamanlar da dahil. Az önce UART örneğinde gösterildiği gibi, bir kaynağı tutarsız bir durumda bırakmak veri bozulmasına yol açabilir.

• İşbirlikçi çizgeleyiciyi kullandığınızda, başka bir göreve geçişin ne zaman gerçekleşeceğini siz kontrol edersiniz. Dolayısıyla bir kaynak tutarsız bir durumdayken başka bir göreve geçişin gerçekleşmemesini sağlayabilirsiniz.

• Yukarıdaki UART örneğinde, Görev 1'in tüm dizesini UART'a yazmayı bitirene kadar *Çalışıyor* (Running) durumundan çıkmamasını sağlayabilir ve bu sayede dizenin başka bir görevin faaliyetleri tarafından bozulma olasılığını ortadan kaldırabilirsiniz.

Şekil 4.22'de gösterildiği gibi, işbirlikçi çizgeleyiciyi kullanmak, sistemleri önleyici çizgeleyiciyi kullanmaya göre daha az duyarlı (responsive) yapar:

• Önleyici çizgeleyiciyi kullandığınızda, çizgeleyici görev en yüksek öncelikli *Hazır* (Ready) durumundaki görev olduğu anda hemen çalıştırmaya başlar. Bu, belirli bir süre içinde yüksek öncelikli olaylara yanıt vermesi gereken gerçek zamanlı sistemlerde genellikle temel bir gerekliliktir.

• İşbirlikçi çizgeleyiciyi kullandığınızda, en yüksek öncelikli *Hazır* (Ready) durumundaki görev haline gelmiş bir göreve geçiş, *Çalışıyor* (Running) durumundaki görev *Engellenmiş* (Blocked) durumuna girene veya taskYIELD() çağırana kadar gerçekleştirilmez.

5 Kuyruk Yönetimi (Queue Management)

5.1 Giriş

'Kuyruklar' (Queues), görevden göreve (task-to-task), görevden kesmeye (task-to-interrupt) ve kesmeden göreve (interrupt-to-task) bir iletişim mekanizması sağlar.

5.1.1 Kapsam

Bu bölüm şunları kapsamaktadır:

- Bir kuyruğun nasıl oluşturulacağı.
- Bir kuyruğun içerdiği verileri nasıl yönettiği.
- Bir kuyruğa veri nasıl gönderilir.
- Bir kuyruktan veri nasıl alınır.
- Bir kuyrukta engellenmiş (block) olmanın ne anlama geldiği.
- Birden fazla kuyrukta nasıl engelleneceği.
- Kuyruktaki verilerin üzerine nasıl yazılacağı (overwrite).
- Kuyruğun nasıl temizleneceği (clear).
- Bir kuyruğa yazarken ve kuyruktan okurken görev önceliklerinin etkisi.

Bu bölüm yalnızca görevden göreve iletişimi kapsar. Bölüm 7 görevden kesmeye ve kesmeden göreve iletişimi kapsamaktadır.

5.2 Bir Kuyruğun Özellikleri

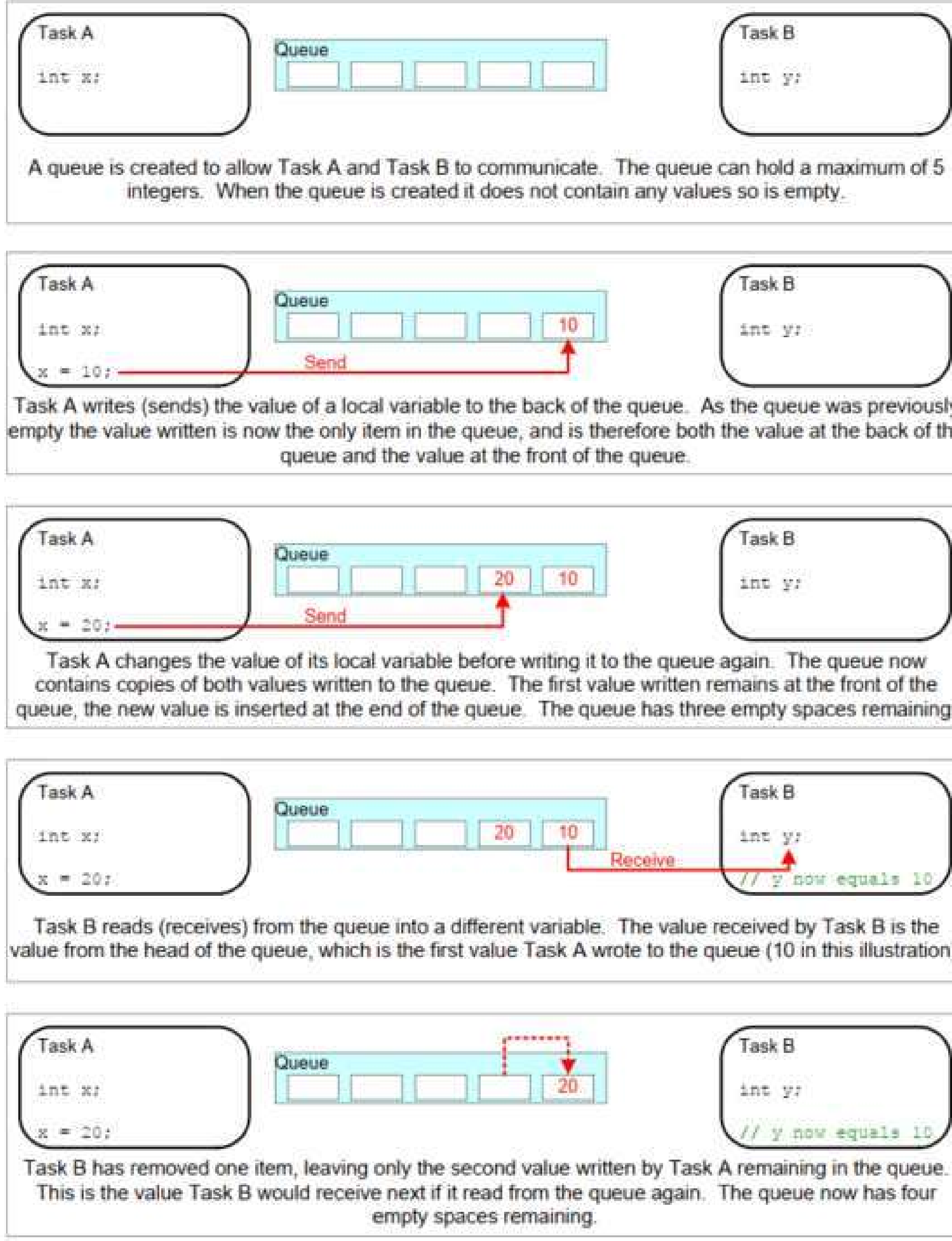
5.2.1 Veri Depolama

Bir kuyruk, sınırlı sayıda sabit boyutlu veri öğesini tutabilir. Bir kuyruğun tutabileceği maksimum öğe sayısına kuyruğun 'uzunluğu' (length) denir. Hem uzunluk hem de her bir veri öğesinin boyutu kuyruk oluşturulurken belirlenir.

(Not: Bölüm TBD'de açıklanan FreeRTOS mesaj arabellekleri - message buffers - değişken uzunluklu mesajları tutan kuyruklara daha hafif bir alternatif sunar.)

Kuyruklar normalde verilerin kuyruğun sonuna (kuyruğuna - tail) yazıldığı ve kuyruğun önünden (başından - head) çıkarıldığı İlk Giren İlk Çıkar (First In First Out - FIFO) arabellekleri olarak kullanılır. Şekil 5.1, FIFO olarak kullanılan bir kuyruğa veri

yazıldığını ve kuyruktan veri okunduğunu göstermektedir. Ayrıca kuyruğun önüne (front) yazmak ve kuyruğun önünde bulunan verilerin üzerine yazmak da mümkündür.



Şekil 5.1 Bir kuyruğa yazma ve kuyruktan okuma örnek sırası

Kuyruk davranışının (queue behaviour) uygulanabileceği iki yol vardır:

1. **Kopya ile Kuyruğa Alma (Queue by copy):** Kopya yoluyla kuyruğa alma, kuyruğa gönderilen verilerin bayt bayt kuyruğa kopyalanması anlamına gelir.

2. **Referans ile Kuyruğa Alma (Queue by reference):** Referans yoluyla kuyruğa alma, kuyruğun, kuyruğa gönderilen verinin kendisini değil, yalnızca ona işaret eden işaretçileri (pointers) tutması anlamına gelir.

FreeRTOS kopya (copy) ile kuyruğa alma yöntemini kullanır, çünkü referansla kuyruğa almaya göre hem daha güçlü hem de kullanımı daha basittir çünkü:

- Kopya ile kuyruğa alma, kuyruğun referans ile kuyruğa almak için kullanılmasına da engel olmaz. Örneğin, kuyruğa alınan verinin boyutu veriyi kuyruğa kopyalamayı pratik olmaktan çıkardığında, verinin kendisi yerine veriye bir işaretçi kuyruğa kopyalanabilir.
- İçinde bildirildiği işlevden çıkıldıktan sonra değişken mevcut olmayacak olsa bile, bir yığın (stack) değişkeni doğrudan bir kuyruğa gönderilebilir.
- Veriler önce verileri tutacak bir arabellek (buffer) tahsis edilmeden bir kuyruğa gönderilebilir—daha sonra verileri ayrılan arabelleğe kopyalar ve arabelleğe bir referansı kuyruğa alırsınız.
- Gönderen görev, kuyruğa gönderilen değişkeni veya arabelleği anında yeniden kullanabilir.
- Gönderen görev ve alan görev tamamen birbirinden ayrılmıştır (de-coupled); bir uygulama tasarımcısının hangi görevin veriye 'sahip' olduğuyla veya veriyi serbest bırakmaktan (releasing) hangi görevin sorumlu olduğuyla ilgilenmesi gerekmez.
- RTOS verileri depolamak için kullanılan belleğin tahsis edilmesinde (allocating memory) tüm sorumluluğu üstlenir.
- Bellek korumalı (Memory protected) sistemler RAM'e erişimi sınırlar, bu durumda referans ile kuyruğa alma işlemi yalnızca gönderen ve alan görevlerin her ikisi de başvurulan verilere erişebilirse gerçekleştirilebilir. Kopya ile kuyruğa alma verilerin bellek koruma sınırlarını (boundaries) geçmesine izin verir.

5.2.2 Birden Fazla Görev (Task) Tarafından Erişim

Kuyruklar başlı başına nesnelerdir (objects) ve varlıklarını bilen herhangi bir görev veya Kesme Servis Rutini (ISR) tarafından erişilebilirler. Herhangi bir sayıda görev aynı kuyruğa yazabilir ve herhangi bir sayıda görev aynı kuyruktan okuyabilir. Uygulamada (pratikte), bir kuyruğun birden fazla yazıcısı olması çok yaygındır, ancak bir kuyruğun birden fazla okuyucusu olması çok daha az yaygındır.

5.2.3 Kuyruk Okumalarında Engelleme (Blocking on Queue Reads)

Bir görev bir kuyruktan veri okumaya çalıştığında isteğe bağlı olarak bir 'blok' süresi (block time) belirtebilir. Bu, kuyruk zaten boşsa görevin kuyruktan verinin alınabilir olmasını (data to become available) beklemek üzere Engellenmiş durumunda tutulduğu süredir.

Kuyruktan veri alınabilmesi için Engellenmiş durumunda bekleyen bir görev, başka bir görev veya kesme kuyruğa veri yerleştirdiğinde otomatik olarak Hazır (Ready)

durumuna geçer. Belirtilen blok süresi, veri elde edilebilir hale gelmeden önce sona ererse, görev Engellenmiş durumundan Hazır durumuna otomatik olarak geçirilir.

Kuyrukların birden çok okuyucusu olabilir, bu nedenle tek bir kuyruğun verileri bekleyen üzerinde engellenmiş birden çok görevi olması mümkündür. Böyle bir durumda, veri elde edilebilir olduğunda yalnızca tek bir görevin engeli kaldırılır (unblocked). Engeli kaldırılan görev, veriyi bekleyenler arasında her zaman en yüksek öncelikli görevdir. Engellenmiş olmuş iki veya daha fazla görev eşit önceliğe sahipse, engeli kaldırılan görev en uzun süredir beklemekte olandır.

5.2.4 Kuyruk Yazmalarında Engelleme (Blocking on Queue Writes)

Bir görev, bir kuyruktan veri okurken yapabileceği gibi bir kuyruğa veri yazarken de isteğe bağlı olarak bir blok süresi belirtebilir. Bu durumda, blok süresi, kuyruğun zaten dolu olması durumunda kuyrukta boş alan (space) oluşmasını beklemek için görevin Engellenmiş durumunda (Blocked state) tutulacağı maksimum süredir.

Kuyrukların birden fazla yazıcısı olabilir, bu nedenle dolu bir kuyruğun bir gönderme (send) işlemini tamamlamak için beklerken üzerinde engellenmiş birden fazla görevi olması mümkündür. Böyle bir durumda, kuyrukta yer açıldığında yalnızca tek bir görevin engeli kaldırılır. Engeli kaldırılan görev, boş alan bekleyenler arasında her zaman en yüksek öncelikli görevdir. Engellenmiş olmuş iki veya daha fazla görev eşit önceliğe sahipse, engeli kaldırılan görev en uzun süredir beklemekte olandır.

5.2.5 Birden Fazla Kuyrukta Engelleme (Blocking on Multiple Queues)

Kuyruklar gruplanarak set (set/küme) haline getirilebilir ve böylece bir görevin setteki (kümedeki) herhangi bir kuyrukta verilerin kullanılabilir hale gelmesini beklemek üzere Engellenmiş durumuna girmesine olanak tanınır. Bölüm 5.6 Birden Fazla Kuyruktan Veri Alma, kuyruk setlerini göstermektedir.

5.2.6 Kuyruklar Oluşturma: Statik ve Dinamik Olarak Tahsis Edilen Kuyruklar

Kuyruklara, `QueueHandle_t` türündeki değişkenler olan tanıtıcılar (handles) aracılığıyla başvurulur. Bir kuyruk kullanılmadan önce açıkça oluşturulmalıdır.

Kuyrukları iki API işlevi oluşturur: `xQueueCreate()`, `xQueueCreateStatic()`.

Her kuyruk, ilki veri yapısını tutmak ve ikincisi kuyruğa alınmış verileri (queued data) tutmak için olmak üzere iki RAM bloğu gerektirir. `xQueueCreate()` gerekli RAM'i yığından (heap) dinamik olarak ayırır. `xQueueCreateStatic()` işleve parametre olarak aktarılan önceden ayrılmış RAM'i kullanır.

5.3 Bir Kuyruğun Kullanımı

5.3.1 xQueueCreate() API İşlevi

Liste 5.1 `xQueueCreate()` işlev prototipini göstermektedir. `xQueueCreateStatic()` işlevinin sırasıyla kuyruğun veri yapısını ve veri depolama alanını tutmak için önceden ayrılmış belleğe işaret eden iki ek parametresi vardır.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Liste 5.1 `xQueueCreate()` API işlev prototipi

`xQueueCreate()` parametreleri ve dönüş değeri:

- **`uxQueueLength`:** Oluşturulan kuyruğun aynı anda tutabileceği maksimum öğe sayısı.
- **uxItemSize:** Kuyrukta saklanabilen her bir veri öğesinin bayt cinsinden boyutu.
- **Dönüş değeri:** `NULL` döndürülürse, kuyruk veri yapılarını ve depolama alanını tahsis etmesi için FreeRTOS'un kullanabileceği yeterli yığın (heap) belleği bulunmadığından kuyruk oluşturulamaz. (Bölüm 2, FreeRTOS yığını hakkında daha fazla bilgi sağlar.) `NULL` olmayan (non-`NULL`) bir değer döndürülürse kuyruk başarıyla oluşturulmuştur ve döndürülen değer oluşturulan kuyruğun tanıtıcısıdır (handle).

Not: `xQueueReset()` önceden oluşturulmuş bir kuyruğu orijinal boş durumuna geri yükleyen bir API işlevidir.

5.3.2 xQueueSendToBack() ve xQueueSendToFront() API İşlevleri

Beklenebileceği gibi `xQueueSendToBack()` bir kuyruğun arkasına (kuyruğuna - tail) veri gönderir ve `xQueueSendToFront()` bir kuyruğun önüne (başına - head) veri gönderir.

`xQueueSend()`, `xQueueSendToBack()` ile eşdeğerdir ve onunla tamamen aynıdır.

Not: `xQueueSendToFront()` veya `xQueueSendToBack()` işlevlerini asla bir kesme servis rutininden (ISR) çağırmayın. Bunların yerine kesme güvenli (interrupt-safe) olan `xQueueSendToFrontFromISR()` ve `xQueueSendToBackFromISR()` sürümleri kullanılmalıdır. Bunlar Bölüm 7'de açıklanmıştır.

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,

                              const void * pvItemToQueue,

                              TickType_t xTicksToWait );
```

Liste 5.2 `xQueueSendToFront()` API işlev prototipi

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
                             const void * pvItemToQueue,
                             TickType_t xTicksToWait );
```

Liste 5.3 `xQueueSendToBack()` API işlev prototipi

`xQueueSendToFront()` ve `xQueueSendToBack()` işlev parametreleri ve dönüş değeri:

- **xQueue:** Verinin gönderildiği (yazıldığı) kuyruğun tanıtıcısı (handle). Kuyruk tanıtıcısı, kuyruğu oluşturmak için kullanılan `xQueueCreate()` veya `xQueueCreateStatic()` çağrısından döndürülmüş olacaktır.
- **`pvItemToQueue`:** Kuyruğa kopyalanacak veriye yönelik bir işaretçi (pointer). Kuyruğun tutabileceği her bir öğenin boyutu kuyruk oluşturulurken belirlenir, böylece o kadar bayt `pvItemToQueue`'dan kuyruk depolama alanına kopyalanır.
- **`xTicksToWait`:** Kuyruğun halihazırda dolu olması durumunda kuyrukta boş yer açılmasını beklemek için görevin Engellenmiş durumunda kalacağı maksimum süre. Hem `xQueueSendToFront()` hem de `xQueueSendToBack()`, `xTicksToWait` sıfırsa ve kuyruk zaten doluysa hemen geri dönecektir (return immediately). Blok süresi tick periyotları cinsinden belirtilir, bu nedenle temsil ettiği mutlak süre tick frekansına bağlıdır. Milisaniye cinsinden belirtilen bir süreyi tick cinsinden belirtilen bir süreye dönüştürmek için `pdMS_TO_TICKS()` makrosu kullanılabilir. `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak, `FreeRTOSConfig.h` içinde `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, görevin süresiz olarak (zaman aşımına uğramadan - without timing out) beklemesine neden olacaktır.
- **Dönüş değeri:** Olası iki dönüş değeri vardır:
    - **`pdPASS`:** Veriler kuyruğa başarıyla gönderildiğinde `pdPASS` döndürülür. Eğer bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse), işlev dönmeden önce kuyrukta yer açılmasını beklemek için çağıran görev Engellenmiş durumuna (Blocked state) yerleştirilmiş, ancak blok süresi sona ermeden önce kuyruğa veri başarıyla yazılmış olabilir.
    - **`errQUEUE_FULL` (`pdFAIL` ile aynı değer):** Kuyruk zaten dolu olduğu için veri kuyruğa yazılamadıysa `errQUEUE_FULL` döndürülür. Eğer bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse) çağıran görev, kuyrukta yer açması için başka bir görevi veya kesmeyi beklemek üzere Engellenmiş durumuna yerleştirilmiş, ancak bu gerçekleşmeden önce belirtilen blok süresi dolmuştur.

5.3.3 xQueueReceive() API İşlevi

`xQueueReceive()` bir kuyruktan bir öğe alır (okur). Alınan öğe kuyruktan kaldırılır.

Not: `xQueueReceive()` işlevini asla bir kesme servis rutininden (ISR) çağırmayın. Kesme güvenli (interrupt-safe) `xQueueReceiveFromISR()` API işlevi Bölüm 7'de açıklanmıştır.

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait );
```

Liste 5.4 `xQueueReceive()` API işlev prototipi

`xQueueReceive()` işlev parametreleri ve dönüş değerleri:

- **xQueue:** Verinin alındığı (okunduğu) kuyruğun tanıtıcısı. Kuyruk tanıtıcısı, kuyruğu oluşturmak için kullanılan `xQueueCreate()` veya `xQueueCreateStatic()` çağrısından döndürülmüş olacaktır.
- **`pvBuffer`:** Alınan verilerin kopyalanacağı belleğe yönelik bir işaretçi. Kuyruğun tuttuğu her bir veri öğesinin boyutu kuyruk oluşturulurken ayarlanır. `pvBuffer` tarafından işaret edilen bellek en az o kadar baytı tutacak kadar büyük olmalıdır.
- **`xTicksToWait`:** Kuyruğun halihazırda boş olması durumunda, kuyrukta veri bulunmasını beklemek üzere görevin Engellenmiş durumunda kalacağı maksimum süre. `xTicksToWait` sıfırsa ve kuyruk zaten boşsa `xQueueReceive()` hemen geri dönecektir. Blok süresi tick periyotları cinsinden belirtilir, bu nedenle temsil ettiği mutlak süre tick frekansına bağlıdır. Milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için `pdMS_TO_TICKS()` makrosu kullanılabilir. `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak, `FreeRTOSConfig.h` içinde `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, görevin süresiz olarak (zaman aşımına uğramadan) beklemesine neden olacaktır.
- **Dönüş değeri:** İki olası dönüş değeri vardır:
  - **`pdPASS`:** Kuyruktan veri başarıyla okunduğunda `pdPASS` döndürülür. Eğer bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse), çağıran görev kuyrukta verinin kullanılabilir olmasını beklemek üzere Engellenmiş durumuna yerleştirilmiş olabilir, ancak veri blok süresi sona ermeden önce kuyruktan başarıyla okunmuştur.
  - **`errQUEUE_EMPTY` (`pdFAIL` ile aynı değer):** Kuyruk zaten boş olduğu için veri okunamadıysa `errQUEUE_EMPTY` döndürülür. Eğer bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse), çağıran görev başka bir görevin veya kesmenin kuyruğa veri göndermesini beklemek üzere Engellenmiş durumuna yerleştirilmiş, ancak bu gerçekleşmeden önce blok süresi dolmuştur.

5.3.4 uxQueueMessagesWaiting() API İşlevi

`uxQueueMessagesWaiting()` halihazırda bir kuyrukta bulunan öğelerin sayısını sorgular.

Not: `uxQueueMessagesWaiting()` işlevini asla bir kesme hizmet rutininden (ISR) çağırmayın. Bunun yerine kesme güvenli olan `uxQueueMessagesWaitingFromISR()` kullanılmalıdır.

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

Liste 5.5 `uxQueueMessagesWaiting()` API işlev prototipi

`uxQueueMessagesWaiting()` işlev parametreleri ve dönüş değeri:

- **xQueue:** Sorgulanmakta olan kuyruğun tanıtıcısı (handle). Kuyruk tanıtıcısı, kuyruğu oluşturmak için kullanılan `xQueueCreate()` veya `xQueueCreateStatic()` çağrısından döndürülmüş olacaktır.
- **Dönüş değeri:** Sorgulanan kuyrukta halihazırda bulunan öğelerin sayısı. Sıfır dönerse kuyruk boştur.

Örnek 5.1 Bir kuyruktan okurken engellenme

Bu örnek bir kuyruk oluşturmayı, birden fazla görevden kuyruğa veri göndermeyi ve kuyruktan veri almayı göstermektedir. Kuyruk `int32_t` türünde veri öğelerini tutmak için oluşturulur. Kuyruğa veri gönderen görevler bir blok süresi (block time) belirtmezken, kuyruktan okuyan görev belirtir.

Kuyruğa gönderim yapan görevler, kuyruktan veri alan görevden daha düşük bir önceliğe sahiptir. Bu, kuyruğun asla birden fazla öğe içermemesi gerektiği anlamına gelir, çünkü kuyruğa veri gönderildiği anda alıcı görevin engeli kaldırılacak, (daha yüksek bir önceliğe sahip olduğu için) gönderici görevi engelleyecek (pre-empt) ve verileri kaldırarak kuyruğu bir kez daha boş bırakacaktır.

Örnek, Liste 5.6'da gösterilen görevin iki örneğini oluşturur; bunlardan biri kuyruğa sürekli olarak 100 değerini yazarken, diğeri aynı kuyruğa sürekli olarak 200 değerini yazar. Görev parametresi (task parameter) bu değerleri her bir görev örneğine aktarmak için kullanılır.

```
static void vSenderTask( void *pvParameters )

{

    int32_t lValueToSend;

    BaseType_t xStatus;



    /*

     * Bu görevin iki örneği oluşturulur, bu nedenle kuyruğa gönderilen değer
görev
```

```
     * parametresi aracılığıyla aktarılır - bu şekilde her örnek farklı bir değer
kullanabilir.

     * Kuyruk int32_t türündeki değerleri tutmak üzere oluşturulmuştur, bu nedenle

     * parametreyi gerekli türe dönüştürün.

     */

    lValueToSend = ( int32_t ) pvParameters;



    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü içinde uygulanır. */

    for( ;; )

    {

        /*

         * Değeri kuyruğa gönderin.

         * İlk parametre verinin gönderileceği kuyruktur. Kuyruk, çizgeleyici
başlatılmadan

         * önce, dolayısıyla bu görev çalışmaya başlamadan önce oluşturulmuştur.

         * İkinci parametre gönderilecek verinin adresidir, bu durumda
lValueToSend'in adresi.

         * Üçüncü parametre Blok süresidir – kuyruğun halihazırda dolu olması
durumunda

         * kuyrukta yer açılmasını beklemek için görevin Engellenmiş durumunda
tutulması

         * gereken süredir. Bu durumda kuyruk hiçbir zaman birden fazla öğe

         * içermemeli ve dolayısıyla hiçbir zaman tam dolu olmamalıdır, bu nedenle
bir blok

         * süresi belirtilmemiştir.

         */

        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );



        if( xStatus != pdPASS )

        {

            /*

             * Gönderme işlemi kuyruk dolu olduğu için tamamlanamadı - kuyruk asla

             * birden fazla öğe içermemesi gerektiğinden bu bir hata olmalıdır!
```

```
            */
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

Liste 5.6 Örnek 5.1'de kullanılan gönderici görevin (sending task) uygulanması

Liste 5.7 kuyruktan veri alan görevin uygulanmasını göstermektedir. Alıcı görev 100 milisaniyelik bir blok süresi (block time) belirler, ardından kuyrukta verinin mevcut olmasını beklemek üzere Engellenmiş durumuna (Blocked state) girer. Kuyrukta veri mevcut olduğunda veya veriler mevcut olmadan 100 milisaniye geçtiğinde Engellenmiş durumundan çıkar. Bu örnekte, kuyruğa sürekli yazan iki görev vardır, bu nedenle 100 milisaniye zaman aşımı (timeout) hiçbir zaman sona ermez.

```
static void vReceiverTask( void *pvParameters )
{
    /* Kuyruktan alınan değerleri tutacak değişkeni tanımlayın. */
    int32_t lReceivedValue;
    BaseType_t xStatus;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* Bu görev de sonsuz bir döngü içinde tanımlanmıştır. */
    for( ;; )
    {
        /*
         * Bu görev kuyruğa yazılan herhangi bir veriyi anında çıkaracağından,
         * bu çağrı her zaman kuyruğu boş bulmalıdır.
         */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }
```

```
        /*
         * Kuyruktan veri alın.
         * İlk parametre verinin alınacağı kuyruktur. Kuyruk çizgeleyici
başlatılmadan önce
         * ve dolayısıyla bu görev ilk kez çalışmadan önce oluşturulur.
         * İkinci parametre alınan verinin yerleştirileceği arabellektir (buffer).
         * Bu durumda arabellek, alınan veriyi tutmak için gerekli boyuta sahip
         * bir değişkenin adresidir.
         * Son parametre blok süresidir - kuyruğun halihazırda boş olması
         * durumunda görevin verilerin elde edilebilir olmasını beklemek
         * için Engellenmiş durumunda kalacağı maksimum süre.
         */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* Veriler kuyruktan başarıyla alındı, alınan değeri yazdırın. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /*
             * 100 ms bekledikten sonra bile kuyruktan veri alınamadı.
             * Gönderici görevler serbest çalıştığı (free running) ve kuyruğa
             * sürekli yazacakları için bu bir hata olmalıdır.
             */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
```

```
}
```

Liste 5.7 Örnek 5.1 için alıcı görevin (receiver task) uygulanması

Liste 5.8 `main()` işlevinin tanımını içermektedir. Bu basitçe çizgeleyiciyi (scheduler) başlatmadan önce kuyruğu ve üç görevi oluşturur. Kuyruk, en fazla beş `int32_t` değeri tutacak şekilde oluşturulmuştur, ancak göreceli görev öncelikleri kuyruğun hiçbir zaman aynı anda birden fazla öğe tutmayacağı anlamına gelir.

```
/*
 * QueueHandle_t türünde bir değişken bildirin. Bu, her üç görev tarafından da
 * erişilen kuyruğun tanıtıcısını (handle) saklamak için kullanılır.
 */
QueueHandle_t xQueue;


int main( void )
{
    /*
     * Kuyruk, her biri int32_t türünde bir değişkeni tutacak kadar büyük olan
     * en fazla 5 değeri tutacak şekilde oluşturulur.
     */
    xQueue = xQueueCreate( 5, sizeof( int32_t ) );


    if( xQueue != NULL )
    {
        /*
         * Kuyruğa veri gönderecek görevin iki örneğini oluşturun.
         * Görev parametresi (task parameter) görevin kuyruğa yazacağı değeri
aktarmak
         * için kullanılır, bu nedenle bir görev kuyruğa sürekli olarak 100
yazarken
         * diğeri sürekli olarak 200 yazacaktır. Her iki görev de
         * 1 önceliğiyle oluşturulur.
         */
```

```
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );


        /*

         * Kuyruktan okuma yapacak görevi oluşturun. Görev öncelik 2 ile,

         * yani gönderici görevlerin önceliğinin üzerinde oluşturulur.

         */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );


        /* Çizgeleyiciyi başlatın, böylece oluşturulan görevler yürütülmeye
başlar. */

        vTaskStartScheduler();

    }

    else

    {

        /* Kuyruk oluşturulamadı. */

    }


    /*

     * Her şey yolundaysa çizgeleyici artık görevleri çalıştıracağından main()

     * buraya asla ulaşmayacaktır. Eğer main() buraya ulaşırsa, boş (idle) görevin

     * oluşturulması için yeterli FreeRTOS yığın (heap) belleği bulunmamış

     * olması muhtemeldir. Bölüm 3, yığın belleği yönetimi hakkında daha fazla

     * bilgi sağlar.

     */

    for( ;; );

}
```

Liste 5.8 Örnek 5.1'deki `main()` işlevinin uygulanması

Şekil 5.2, Örnek 5.1 tarafından üretilen çıktıyı göstermektedir.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
Received =   100
Received =   200
```

Şekil 5.2 Örnek 5.1 yürütüldüğünde üretilen çıktı

Şekil 5.3 yürütme dizisini (sequence of execution) göstermektedir.

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Sender 2 runs after the Receiver has blocked.

3 - The Receiver task empties the queue then enters the Blocked state again. This time Sender 1 runs after the Receiver has blocked.

Receiver

Sender 2

Sender 1

t1

Time

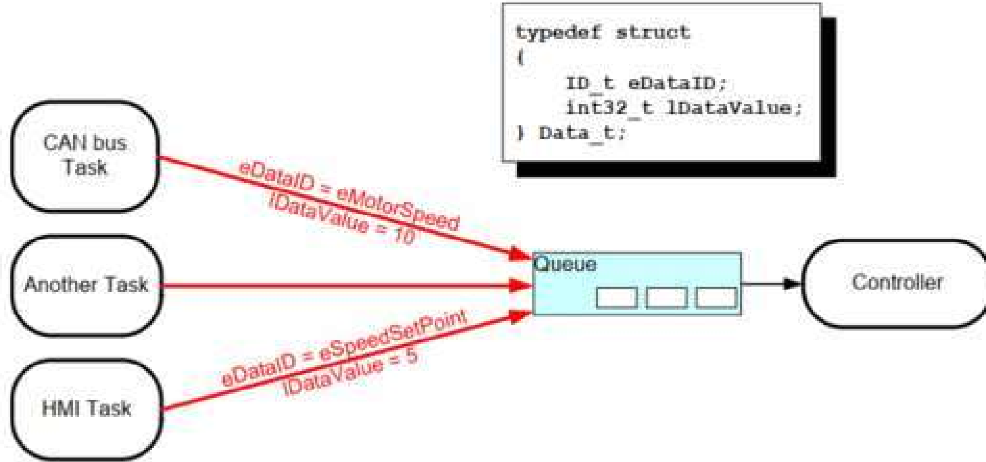
2 - Sender 2 writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ....

Şekil 5.3 Örnek 5.1 tarafından üretilen yürütme dizisi

5.4 Birden Fazla Kaynaktan Veri Alma

FreeRTOS tasarımlarında bir görevin birden fazla kaynaktan veri alması yaygın bir durumdur. Alıcı görevin (receiving task) veriyle ne yapacağını belirlemek için verinin nereden geldiğini bilmesi gerekir. Bunu başarmak için kolay bir tasarım deseni (design pattern), Şekil 5.4'te gösterildiği gibi, hem veri değerini hem de veri kaynağını içeren yapıları (structures) aktarmak için tek bir kuyruk kullanmaktır.



Şekil 5.4 Kuyruk üzerinde yapıların (structures) gönderildiği örnek bir senaryo

Şekil 5.4'e bakıldığında:

- Oluşturulan kuyruk `Data_t` türündeki yapıları (structures) tutar. Bu yapı, hem bir veri değerinin hem de verinin ne anlama geldiğini belirten numaralandırılmış bir türün (enumerated type) kuyruğa tek bir mesajda gönderilmesine olanak tanır.
- Merkezi bir Denetleyici (Controller) görevi birincil sistem işlevini yerine getirir. Bu görevin kuyruk üzerinde kendisine iletilen girdilere ve sistem durumundaki değişikliklere tepki vermesi gerekir.
- Bir CAN veri yolu (bus) görevi, CAN veri yolu arayüzü işlevselliğini sarmalamak (encapsulate) için kullanılır. CAN veri yolu görevi bir mesajı alıp kodunu çözdüğünde (decoded), halihazırda kodu çözülmüş olan mesajı bir `Data_t` yapısı içinde Denetleyici görevine gönderir. Aktarılan yapının `eDataID` üyesi Denetleyici görevine verinin ne olduğunu söyler. Burada gösterilen durumda, bu bir motor hızı değeridir. Aktarılan yapının `lDataValue` üyesi Denetleyici görevine gerçek motor hızı değerini söyler.
- Bir İnsan Makine Arayüzü (Human Machine Interface - HMI) görevi, tüm HMI işlevselliğini sarmalamak (encapsulate) için kullanılır. Makine operatörü muhtemelen komutları girebilir ve HMI görevi içinde algılanması ve yorumlanması gereken bir dizi yolla değerleri sorgulayabilir. Yeni bir komut girildiğinde, HMI görevi komutu bir `Data_t` yapısı içinde Denetleyici görevine gönderir. Aktarılan yapının `eDataID` üyesi Denetleyici görevine verinin ne olduğunu söyler. Burada gösterilen durumda, bu yeni bir ayar noktası (set point) değeridir. Aktarılan yapının `lDataValue` üyesi, Denetleyici görevine gerçek ayar noktası değerini söyler.

Bölüm (RB-TBD) bu tasarım deseninin denetleyici görevinin bir yapıyı (structure) kuyruğa alan göreve doğrudan yanıt verebilmesi için nasıl genişletileceğini göstermektedir.

Örnek 5.2 Bir kuyruğa gönderim yaparken engellenme ve kuyrukta yapı (structure) gönderme

Örnek 5.2, Örnek 5.1'e benzer, ancak görev öncelikleri tersine çevrilmiştir (reversed), böylece alıcı görev gönderici görevlerden daha düşük bir önceliğe sahiptir. Ayrıca, oluşturulan kuyruk tam sayılar (integers) yerine yapılar (structures) tutar.

Liste 5.9, Örnek 5.2 tarafından kullanılan yapının (structure) tanımını göstermektedir.

```
/* Verinin kaynağını tanımlamak için kullanılan numaralandırılmış bir tür (enum)
tanımlayın. */

typedef enum

{

    eSender1,

    eSender2

} DataSource_t;



/* Kuyruktan geçirilecek (pass on) yapıyı (structure) tanımlayın. */

typedef struct

{

    uint8_t ucValue;

    DataSource_t eDataSource;

} Data_t;



/*

 * Biri Görev 1'den, diğeri Görev 2'den geçirilecek şekilde iki xStructsToSend
değişkeni

 * bildirin. Her değişken bir döngü içinde sürekli olarak gönderilir.

 */

static const Data_t xStructsToSend[ 2 ] =

{

    { 100, eSender1 }, /* Görev 1 için kullanılır. */

    { 200, eSender2 }  /* Görev 2 için kullanılır. */

};
```

Liste 5.9 Bir kuyruktan geçirilecek olan yapının tanımı, ayrıca örnekte kullanılmak üzere iki değişkenin bildirimi

Örnek 5.1'de, alıcı görev en yüksek önceliğe sahiptir, bu nedenle kuyruk asla birden fazla öğe içermez. Bu, kuyruğa veri yerleştirilir yerleştirilmez alıcı görevin gönderici görevleri engellemesi (pre-empts) nedeniyle gerçekleşir. Örnek 5.2'de gönderici görevler daha yüksek önceliğe sahiptir, bu nedenle kuyruk normalde dolu olacaktır. Bunun nedeni, alıcı görev kuyruktan bir öğe çıkarır çıkarmaz, kuyruğu anında yeniden dolduran gönderici görevlerden biri tarafından engellenmesidir. Gönderici görev daha sonra kuyrukta yeniden yer açılmasını beklemek üzere tekrar Engellenmiş durumuna (Blocked state) girer.

Liste 5.10 gönderici görevin uygulamasını (implementation) göstermektedir. Gönderici görev 100 milisaniyelik bir blok süresi belirtir, böylece kuyruk her dolduğunda yer açılmasını beklemek için Engellenmiş durumuna girer. Kuyrukta boş yer açıldığında veya yer açılmadan 100 milisaniye geçtiğinde Engellenmiş durumundan çıkar. Bu örnekte, alıcı görev sürekli olarak kuyrukta yer açar, bu nedenle 100 milisaniye zaman aşımı hiçbir zaman sona ermez.

```
static void vSenderTask( void *pvParameters )
{
    BaseType_t xStatus;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );


    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngü içinde uygulanır. */
    for( ;; )
    {
        /*
         * Kuyruğa gönder.
         * İkinci parametre, gönderilen yapının adresidir. Adres görev parametresi
         * olarak geçirilir, bu nedenle doğrudan pvParameters kullanılır.
         * Üçüncü parametre Blok süresidir - kuyruk zaten doluysa kuyrukta boş yer
         * açılmasını beklemek için görevin Engellenmiş durumunda tutulması
gereken süre.
         * Gönderici görevler alıcı görevden daha yüksek önceliğe sahip olduğundan
ve
         * dolayısıyla kuyruğun dolması beklendiğinden bir blok süresi belirtilir.
         * Alıcı görev, yalnızca her iki gönderici görev de Engellenmiş
durumundayken
```

```
         * kuyruktan öğe çıkaracaktır.
         */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {
            /*
             * 100 ms bekledikten sonra bile gönderme işlemi tamamlanamadı.
             * Her iki gönderici görev de Engellenmiş durumuna geçtiğinde alıcı
görev anında
             * kuyrukta yer açması gerektiğinden bu bir hata olmalıdır.
             */
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

Liste 5.10 Örnek 5.2 için gönderici görevin uygulanması

Alıcı görev en düşük önceliğe sahiptir, bu nedenle yalnızca her iki gönderici görev de Engellenmiş durumundayken çalışır. Gönderici görevler yalnızca kuyruk dolduğunda Engellenmiş durumuna girerler, bu nedenle alıcı görev yalnızca kuyruk zaten doluyken yürütülecektir. Bu nedenle, bir blok süresi belirtmese bile her zaman veri almayı bekler. Liste 5.11 alıcı görevin uygulamasını göstermektedir.

```
static void vReceiverTask( void *pvParameters )
{
    /* Kuyruktan alınan değerleri tutacak yapıyı (structure) bildirin. */
    Data_t xReceivedStructure;
    BaseType_t xStatus;

    /* Bu görev de sonsuz bir döngü içinde tanımlanmıştır. */
    for( ;; )
```

```
    {
        /*
         * En düşük önceliğe sahip olduğu için bu görev yalnızca gönderici
görevler
         * Engellenmiş durumundayken çalışacaktır. Gönderici görevler yalnızca
kuyruk
         * dolduğunda Engellenmiş durumuna girecektir, bu nedenle bu görev her
zaman
         * kuyruktaki öğe sayısının bu durumda 3 olan kuyruk uzunluğuna (queue
length)
         * eşit olmasını bekler.
         */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /*
         * Kuyruktan veri alın.
         * İkinci parametre alınan verinin yerleştirileceği arabellektir. Bu
durumda
         * arabellek basitçe, alınan yapıyı (structure) tutmak için gerekli boyuta
         * sahip bir değişkenin adresidir.
         * Son parametre blok süresidir - kuyruk zaten boşsa verilerin mevcut
         * olmasını beklemek için görevin Engellenmiş durumunda kalacağı maksimum
süre.
         * Bu durumda bu görev yalnızca kuyruk dolduğunda çalışacağından bir
         * blok süresine gerek yoktur.
         */
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
```

```
            /* Veri kuyruktan başarıyla alındı, alınan değeri ve kaynağını
yazdırın. */

            if( xReceivedStructure.eDataSource == eSender1 )

            {

                vPrintStringAndNumber( "From Sender 1 = ",
xReceivedStructure.ucValue );

            }

            else

            {

                vPrintStringAndNumber( "From Sender 2 = ",
xReceivedStructure.ucValue );

            }

        }

        else

        {

            /*

             * Kuyruktan hiçbir şey alınmadı. Bu görev yalnızca kuyruk dolduğunda

             * çalışması gerektiğinden bu bir hata olmalıdır.

             */

            vPrintString( "Could not receive from the queue.\r\n" );

        }

    }

}
```

Liste 5.11 Örnek 5.2 için alıcı görevin tanımı

`main()` işlevi önceki örnekten yalnızca biraz farklıdır. Kuyruk üç `Data_t` yapısını tutacak şekilde oluşturulur ve gönderici ve alıcı görevlerin öncelikleri tersine çevrilir. Liste 5.12 `main()` işlevinin uygulamasını göstermektedir.

```
int main( void )

{

    /* Kuyruk, Data_t türünde en fazla 3 yapıyı (structure) tutacak şekilde
oluşturulur. */
```

```
    xQueue = xQueueCreate( 3, sizeof( Data_t ) );


    if( xQueue != NULL )

    {

        /*

         * Kuyruğa yazacak görevin iki örneğini oluşturun. Parametre, görevin

         * kuyruğa yazacağı yapıyı geçirmek için kullanılır, böylece bir görev
kuyruğa

         * sürekli olarak xStructsToSend[ 0 ] gönderirken, diğeri sürekli olarak

         * xStructsToSend[ 1 ] gönderir. Her iki görev de alıcının önceliğinin

         * üzerinde olan 2 önceliğiyle (priority 2) oluşturulur.

         */

        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2,
NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2,
NULL );


        /*

         * Kuyruktan okuma yapacak görevi oluşturun. Görev öncelik 1 ile, yani
gönderici

         * görevlerin önceliğinin altında oluşturulur.

         */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );


        /* Çizgeleyiciyi başlatın, böylece oluşturulan görevler yürütülmeye
başlar. */

        vTaskStartScheduler();

    }

    else

    {

        /* Kuyruk oluşturulamadı. */

    }
```

```
    /*
     * Her şey yolundaysa çizgeleyici artık görevleri çalıştıracağından main()
     * buraya asla ulaşmayacaktır. Eğer main() buraya ulaşırsa, boş (idle) görevin
     * oluşturulması için yeterli yığın (heap) belleği bulunmamış olması
muhtemeldir.
     */
    for( ;; );
}
```

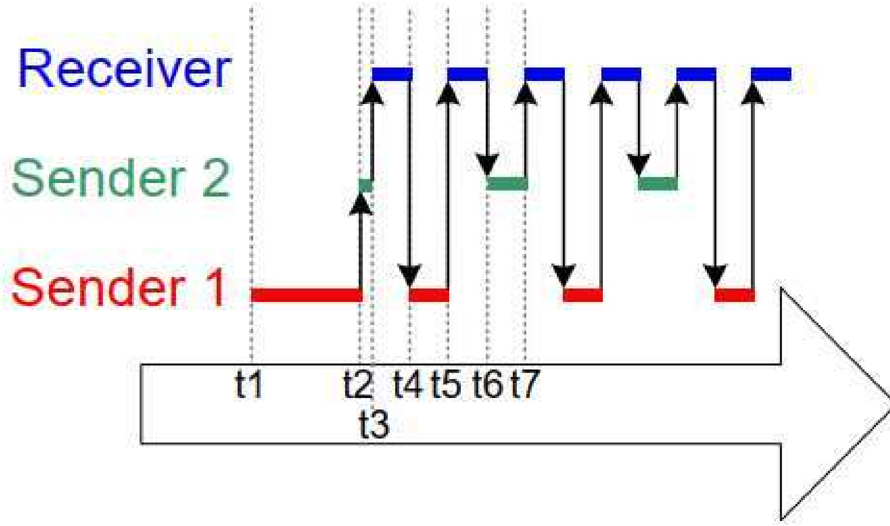
Liste 5.12 Örnek 5.2 için `main()` işlevinin uygulanması

Şekil 5.5, Örnek 5.2 tarafından üretilen çıktıyı göstermektedir.

```
C:\Windows\system32\cmd.exe
C:\temp>Example011
From Sender 1 =  100
From Sender 1 =  100
From Sender 1 =  100
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
From Sender 2 =  200
From Sender 1 =  100
```

Şekil 5.5 Örnek 5.2 tarafından üretilen çıktı

Şekil 5.6, gönderici görevlerin önceliğinin alıcı görevin önceliğinden yüksek olmasından kaynaklanan yürütme dizisini (sequence of execution) göstermektedir. Şekil 5.6'nın daha ayrıntılı bir açıklaması ve ilk dört mesajın neden aynı görevden geldiğinin açıklaması aşağıda verilmiştir.



Şekil 5.6 Örnek 5.2 tarafından üretilen yürütme dizisi

Şekil 5.6'nın Açıklaması:

- **t1:** Gönderici 1 (Sender 1) görevi yürütülür ve kuyruğa 3 veri öğesi gönderir.
- **t2:** Kuyruk dolar, bu nedenle Gönderici 1 bir sonraki gönderiminin tamamlanmasını beklemek üzere Engellenmiş durumuna (Blocked state) girer. Gönderici 2 (Sender 2) görevi artık çalışabilecek en yüksek öncelikli görevdir, bu nedenle Çalışıyor (Running) durumuna girer.
- **t3:** Gönderici 2 görevi kuyruğun zaten dolu olduğunu anlar, bu nedenle ilk gönderiminin tamamlanmasını beklemek üzere Engellenmiş durumuna girer. Alıcı (Receiver) görevi artık çalışabilecek en yüksek öncelikli görevdir, bu nedenle Çalışıyor durumuna girer.
- **t4:** Alıcı görevin önceliğinden daha yüksek önceliğe sahip iki görev kuyrukta yer açılmasını beklemektedir, bu da Alıcı görevinin kuyruktan bir öğeyi çıkarır çıkarmaz engellenmesine (pre-empted) neden olur. Gönderici 1 ve Gönderici 2 görevleri aynı önceliğe sahiptir, bu nedenle çizgeleyici (scheduler) Çalışıyor durumuna girecek görev olarak en uzun süredir bekleyen görevi seçer; bu durumda bu görev Gönderici 1'dir.
- **t5:** Gönderici 1 görevi kuyruğa başka bir veri öğesi daha gönderir. Kuyrukta yalnızca bir boş yer vardır, bu nedenle Gönderici 1 görevi bir sonraki gönderiminin tamamlanmasını beklemek için Engellenmiş durumuna girer. Alıcı görev yine çalışabilen en yüksek öncelikli görevdir, bu nedenle Çalışıyor durumuna girer. Gönderici 1 görevi şimdi kuyruğa dört öğe göndermiştir ve Gönderici 2 görevi hala kuyruğa ilk öğesini göndermeyi beklemektedir.
- **t6:** Alıcı görevin önceliğinden daha yüksek önceliğe sahip iki görev kuyrukta yer açılmasını beklemektedir, bu nedenle Alıcı görev kuyruktan bir öğe çıkarır çıkarmaz engellenir (pre-empted). Bu kez Gönderici 2 Gönderici 1'den daha uzun süredir beklemektedir, bu nedenle Gönderici 2 Çalışıyor durumuna girer.
- **t7:** Gönderici 2 görevi kuyruğa bir veri öğesi gönderir. Kuyrukta yalnızca bir boş yer kalmıştır, bu nedenle Gönderici 2 bir sonraki gönderiminin

tamamlanmasını beklemek için Engellenmiş durumuna girer. Hem Gönderici 1 hem de Gönderici 2 görevleri kuyrukta yer açılmasını beklemektedir, bu nedenle Çalışıyor durumuna girebilecek tek görev Alıcı görevidir.

5.5 Büyük veya Değişken Boyutlu Verilerle Çalışma

5.5.1 İşaretçileri (Pointers) Kuyruğa Alma

Kuyrukta saklanan verinin boyutu büyükse, verinin kendisini bayt bayt kuyruğa ve kuyruktan kopyalamak yerine, veriye işaret eden işaretçileri (pointers) aktarmak (transfer etmek) için kuyruğu kullanmak tercih edilir. İşaretçileri aktarmak hem işlem süresi açısından hem de kuyruğu oluşturmak için gereken RAM miktarı açısından daha verimlidir. Ancak, işaretçileri kuyruğa alırken (queuing pointers) şunlardan emin olmak için çok dikkatli olunmalıdır:

1. **İşaret edilen RAM'in sahibinin açıkça tanımlanması:** Bellek bir işaretçi (pointer) aracılığıyla görevler arasında paylaşılırken, her iki görevin de bellek içeriğini aynı anda (simultaneously) değiştirmemesini veya bellek içeriğinin geçersiz (invalid) veya tutarsız (inconsistent) olmasına neden olabilecek herhangi bir eylemde bulunmamasını sağlamak çok önemlidir. İdeal olarak, işaretçi kuyruğa gönderilmeden önce yalnızca gönderici görevin belleğe erişmesine izin verilmeli ve işaretçi kuyruktan alındıktan sonra yalnızca alıcı görevin belleğe erişmesine izin verilmelidir.
2. **İşaret edilen RAM'in geçerli kalması:** Eğer işaret edilen bellek dinamik olarak tahsis edilmişse (allocated dynamically) veya önceden tahsis edilmiş arabelleklerden (pre-allocated buffers) oluşan bir havuzdan elde edilmişse, belleği serbest bırakmaktan (freeing the memory) tam olarak bir görev sorumlu olmalıdır. Hiçbir görev, serbest bırakıldıktan sonra belleğe erişmeye çalışmamalıdır.

Bir işaretçi, bir görev yığınında (task stack) tahsis edilmiş olan verilere erişmek için asla kullanılmamalıdır. Veriler yığın çerçevesi (stack frame) değiştikten sonra geçerli olmayacaktır.

Örnek olarak Liste 5.13, 5.14 ve 5.15, bir arabelleğe (buffer) yönelik bir işaretçiyi bir görevden diğerine göndermek için bir kuyruğun nasıl kullanılacağını göstermektedir:

- Liste 5.13, en fazla 5 işaretçi tutabilen bir kuyruk oluşturur.
- Liste 5.14 bir arabellek ayırır, arabelleğe bir dize (string) yazar, ardından arabelleğe bir işaretçiyi kuyruğa gönderir.
- Liste 5.15 kuyruktan bir arabelleğe giden bir işaretçi alır, ardından arabellekte bulunan dizeyi yazdırır.

```
/* Oluşturulmakta olan kuyruğun tanıtıcısını tutmak için QueueHandle_t türünde

 * bir değişken bildirin. */
```

```
QueueHandle_t xPointerQueue;


/* Bu durumda karakter işaretçileri olmak üzere en fazla 5 işaretçiyi tutabilen
 * bir kuyruk oluşturun. */
xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

Liste 5.13 İşaretçileri (pointers) tutan bir kuyruk oluşturma

```
/*
 * Bir arabellek elde eden, arabelleğe bir dize yazan, ardından arabelleğin
 * adresini Liste 5.13'te oluşturulan kuyruğa gönderen bir görev.
 */
void vStringSendingTask( void *pvParameters )
{
    char *pcStringToSend;
    const size_t xMaxStringLength = 50;
    BaseType_t xStringNumber = 0;


    for( ;; )
    {
        /*
         * En az xMaxStringLength karakter büyüklüğünde bir arabellek elde edin.
         * prvGetBuffer() işlevinin uygulaması gösterilmemiştir – arabelleği
önceden
         * tahsis edilmiş arabellekler havuzundan elde edebilir veya arabelleği
         * sadece dinamik olarak ayırabilir.
         */
        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );


        /* Arabelleğe bir dize yazın. */
```

```
        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
xStringNumber );


        /* Sayacı artırın, böylece bu görevin her yinelemesinde dize farklı olur.
*/
        xStringNumber++;


        /*
         * Arabelleğin adresini Liste 5.13'te oluşturulan kuyruğa gönderin.
         * Arabelleğin adresi pcStringToSend değişkeninde saklanır.
         */
        xQueueSend( xPointerQueue,
                    /* Kuyruğun tanıtıcısı. */
                    &pcStringToSend, /* Arabelleği işaret eden işaretçinin adresi.
*/
                    portMAX_DELAY );
    }
}
```

Liste 5.14 Bir arabelleğe bir işaretçi göndermek için bir kuyruk kullanılması

```
/*
 * Liste 5.13'te oluşturulan ve Liste 5.14'te yazılan kuyruktan bir arabelleğin
 * adresini alan görev. Arabellek, yazdırılan bir dize içerir.
 */
void vStringReceivingTask( void *pvParameters )
{
    char *pcReceivedString;

    for( ;; )
    {
        /* Bir arabelleğin adresini alın. */
```

```
        xQueueReceive( xPointerQueue,

                       /* Kuyruğun tanıtıcısı. */

                       &pcReceivedString, /* Arabelleğin adresini
pcReceivedString'de saklayın. */

                       portMAX_DELAY );


        /* Arabellek bir dize tutar, onu yazdırın. */

        vPrintString( pcReceivedString );


        /*

         * Arabelleğe artık ihtiyaç yoktur - serbest bırakılabilmesi (freed)

         * veya yeniden kullanılabilmesi için serbest bırakın (release).

         */

        prvReleaseBuffer( pcReceivedString );

    }

}
```

Liste 5.15 Bir arabelleğe işaretçi almak için bir kuyruk kullanılması

5.5.2 Farklı Türlerde ve Uzunluklarda Veri Göndermek İçin Bir Kuyruk Kullanma

(Not: FreeRTOS mesaj arabellekleri, değişken uzunluklu verileri tutan kuyruklara daha hafif bir alternatiftir.)

Bu kitabın önceki bölümlerinde iki güçlü tasarım deseni gösterilmiştir; bir kuyruğa yapı (structure) göndermek ve bir kuyruğa işaretçi (pointer) göndermek. Bu tekniklerin birleştirilmesi, bir görevin herhangi bir veri kaynağından herhangi bir veri türünü almak için tek bir kuyruk kullanmasına olanak tanır. FreeRTOS+TCP TCP/IP yığınının (stack) uygulanması bunun nasıl başarıldığına dair pratik bir örnek sunar.

Kendi görevinde çalışan TCP/IP yığını, birçok farklı kaynaktan gelen olayları işlemelidir. Farklı olay türleri, farklı veri türleri ve uzunlukları ile ilişkilendirilir. `IPStackEvent_t` yapıları TCP/IP görevinin dışında meydana gelen tüm olayları açıklar ve TCP/IP görevine bir kuyruk üzerinde gönderilir. Liste 5.16 `IPStackEvent_t` yapısını göstermektedir. `IPStackEvent_t` yapısının `pvData` üyesi, doğrudan bir değeri tutmak veya bir arabelleği (buffer) işaret etmek için kullanılabilen bir işaretçidir.

```
/* Olayları tanımlamak için TCP/IP yığınında kullanılan numaralandırılmış
```

```
 * (enumerated) türlerin bir alt kümesi. */

typedef enum

{

    eNetworkDownEvent = 0,  /* Ağ arayüzü kayboldu veya (yeniden) bağlanması
gerekiyor. */

    eNetworkRxEvent,        /* Ağdan bir paket alındı. */

    eTCPAcceptEvent,        /* FreeRTOS_accept() yeni bir istemciyi kabul etmek

                             * veya beklemek için çağrıldı. */

    /* Diğer olay türleri burada görünür ancak bu listede gösterilmemiştir. */

} eIPEvent_t;



/* Olayları tanımlayan ve TCP/IP görevine bir kuyrukta gönderilen yapı. */

typedef struct IP_TASK_COMMANDS

{

    /* Olayı tanımlayan numaralandırılmış bir tür. Yukarıdaki eIPEvent_t tanımına
bakın. */

    eIPEvent_t eEventType;



    /* Bir değeri tutabilen veya bir arabelleği işaret edebilen genel (generic)
bir işaretçi. */

    void *pvData;

} IPStackEvent_t;
```

Liste 5.16 FreeRTOS+TCP'de TCP/IP yığın görevine olay göndermek için kullanılan yapı

Örnek TCP/IP olayları ve bunlarla ilişkili veriler şunları içerir:

- **`eNetworkRxEvent`:** Ağdan bir veri paketi alındı.
  Ağ arabirimi (network interface), `IPStackEvent_t` türünde bir yapı kullanarak veri alındı olaylarını TCP/IP görevine gönderir. Yapının `eEventType` üyesi `eNetworkRxEvent` olarak ayarlanır ve yapının `pvData` üyesi alınan verileri içeren arabelleği işaret etmek için kullanılır. Liste 5.17, sözde (pseudo) kod örneğini göstermektedir.

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t *pxRxedData )
```

```
{
    IPStackEvent_t xEventStruct;

    /* IPStackEvent_t yapısını tamamlayın. Alınan veri pxRxedData'da saklanır. */
    xEventStruct.eEventType = eNetworkRxEvent;
    xEventStruct.pvData = ( void * ) pxRxedData;

    /* IPStackEvent_t yapısını TCP/IP görevine gönder. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

Liste 5.17 Ağdan alınan verileri TCP/IP görevine göndermek için bir `IPStackEvent_t` yapısının nasıl kullanıldığını gösteren sözde (pseudo) kod

- **`eTCPAcceptEvent`:** Bir soket (socket) bir istemciden gelen bir bağlantıyı kabul edecek veya bekleyecektir.
  `FreeRTOS_accept()`'i çağıran görev, `IPStackEvent_t` türünde bir yapı kullanarak TCP/IP görevine kabul (accept) olaylarını gönderir. Yapının `eEventType` üyesi `eTCPAcceptEvent` olarak ayarlanır ve yapının `pvData` üyesi bağlantı kabul eden soketin tanıtıcısına (handle) ayarlanır. Liste 5.18 sözde kod örneğini göstermektedir.

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )
{
    IPStackEvent_t xEventStruct;

    /* IPStackEvent_t yapısını tamamlayın. */
    xEventStruct.eEventType = eTCPAcceptEvent;
    xEventStruct.pvData = ( void * ) xSocket;

    /* IPStackEvent_t yapısını TCP/IP görevine gönderin. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

Liste 5.18 TCP/IP görevine bağlantı kabul eden bir soketin tanıtıcısını göndermek için `IPStackEvent_t` yapısının nasıl kullanıldığını gösteren sözde kod

- **`eNetworkDownEvent`:** Ağın bağlanması veya yeniden bağlanması gerekir. Ağ arayüzü `IPStackEvent_t` türünde bir yapı kullanarak ağ kapalı (network down) olaylarını TCP/IP görevine gönderir. Yapının `eEventType` üyesi `eNetworkDownEvent` olarak ayarlanır. Ağın kesilmesi olayları herhangi bir veriyle ilişkili değildir, bu nedenle yapının `pvData` üyesi kullanılmaz. Liste 5.19 sözde kod örneğini göstermektedir.

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )

{

    IPStackEvent_t xEventStruct;



    /* IPStackEvent_t yapısını tamamlayın. */

    xEventStruct.eEventType = eNetworkDownEvent;

    xEventStruct.pvData = NULL; /* Kullanılmıyor, ancak eksiksiz olması için NULL
olarak ayarlandı. */



    /* IPStackEvent_t yapısını TCP/IP görevine gönder. */

    xSendEventStructToIPTask( &xEventStruct );

}
```

Liste 5.19 Ağ çöktü (network down) olayını TCP/IP görevine göndermek için `IPStackEvent_t` yapısının nasıl kullanıldığını gösteren sözde kod

Liste 5.20, TCP/IP görevi içindeki bu olayları alan ve işleyen kodu göstermektedir. Kuyruktan alınan `IPStackEvent_t` yapılarının `eEventType` üyesinin, `pvData` üyesinin nasıl yorumlanacağını belirlemek için kullanıldığı görülebilir.

```
IPStackEvent_t xReceivedEvent;



/*

 * Bir olay alınana veya olay alınmadan xNextIPSleep keneleri (ticks) geçene kadar

 * ağ olay kuyruğunda engellen. Bir olayın alınması yerine zaman aşımına (timed
out)

 * uğradığı için xQueueReceive() işlevinden geri dönülmesi ihtimaline karşı
```

```
 * eEventType, eNoEvent olarak ayarlanır.

 */

xReceivedEvent.eEventType = eNoEvent;

xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );


/* Varsa, hangi olay alındı? */

switch( xReceivedEvent.eEventType )

{

    case eNetworkDownEvent :

        /* Bir bağlantı (yeniden) kurmaya çalışın. Bu olay herhangi bir veriyle

         * ilişkili değildir. */

        prvProcessNetworkDownEvent();

        break;


    case eNetworkRxEvent:

        /* Ağ arayüzü yeni bir paket aldı. Alınan veriye bir işaretçi, alınan

         * IPStackEvent_t yapısının pvData üyesinde depolanır. Alınan veriyi

         * işleyin. */

        prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * ) (
xReceivedEvent.pvData ) );

        break;


    case eTCPAcceptEvent:

        /* FreeRTOS_accept() API işlevi çağrıldı. Bağlantıyı kabul eden

         * soketin tanıtıcısı, alınan IPStackEvent_t yapısının pvData

         * üyesinde saklanır. */

        xSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );

        xTCPCheckNewClient( xSocket );

        break;
```

```
    /* Diğer olay türleri de aynı şekilde işlenir ancak burada gösterilmemiştir.
*/

}
```

Liste 5.20 Bir `IPStackEvent_t` yapısının nasıl alındığını ve işlendiğini gösteren sözde kod

5.6 Birden Fazla Kuyruktan Veri Alma

5.6.1 Kuyruk Setleri (Queue Sets)

Çoğu zaman uygulama tasarımları, tek bir görevin farklı boyutlarda, farklı anlamlara sahip ve farklı kaynaklardan gelen verileri almasını gerektirir. Önceki bölüm, yapıları (structures) alan tek bir kuyruk kullanarak bunun düzgün ve verimli bir şekilde nasıl yapılacağını göstermiştir. Bununla birlikte, bazen bir uygulamanın tasarımcısı tasarım seçeneklerini sınırlayan kısıtlamalarla çalışır ve bu da bazı veri kaynakları için ayrı bir kuyruk kullanımını zorunlu kılar. Örneğin, bir tasarıma entegre edilen üçüncü taraf (third party) kodlar, özel (dedicated) bir kuyruğun varlığını varsayabilir. Bu gibi durumlarda 'kuyruk seti' (queue set) kullanılabilir.

Kuyruk setleri, görevin hangisinin veri içerdiğini belirlemek için her kuyruğu sırayla yoklamadan (polling yapmadan), birden fazla kuyruktan veri almasına olanak tanır.

Birden fazla kaynaktan veri almak için bir kuyruk seti kullanan bir tasarım, yapıları (structures) alan tek bir kuyruk kullanarak aynı işlevselliği elde eden bir tasarımdan daha az düzenli ve daha az verimlidir. Bu nedenle, yalnızca tasarım kısıtlamaları (constraints) kullanımlarını kesinlikle gerekli kılıyorsa kuyruk setlerinin kullanılması önerilir.

Aşağıdaki bölümlerde bir kuyruk setinin nasıl kullanılacağı açıklanmaktadır:

1. Bir kuyruk seti oluşturulur.
2. Kuyruklar kümeye (set) eklenir. (Semaforlar da bir kuyruk kümesine eklenebilir. Semaforlar bu kitabın ilerleyen bölümlerinde açıklanmıştır.)
3. Set içindeki hangi kuyrukların veri içerdiğini belirlemek için kuyruk setinden okuma yapılır. Bir setin üyesi olan bir kuyruk veri aldığında, alıcı kuyruğun tanıtıcısı (handle) kuyruk setine gönderilir ve bir görev kuyruk setinden okuma yapan bir işlevi çağırdığında geri döndürülür. Bu nedenle, bir kuyruk setinden bir kuyruk tanıtıcısı döndürülürse, tanıtıcı tarafından başvurulan kuyruğun veri içerdiği bilinir ve görev doğrudan o kuyruktan okuyabilir.

Not: Bir kuyruk bir kuyruk setinin üyesiyse, tanıtıcısı (handle) kuyruk setinden her alındığında o kuyruktan okuma yapmanız gerekir ve tanıtıcısı kuyruk setinden alınmadan önce kuyruktan okuma yapmamalısınız.

Kuyruk seti işlevi, `FreeRTOSConfig.h`'de `configUSE_QUEUE_SETS` derleme zamanı yapılandırma sabitinin 1 olarak ayarlanmasıyla etkinleştirilir.

5.6.2 xQueueCreateSet() API İşlevi

Bir kuyruk seti (queue set) kullanılmadan önce açıkça (explicitly) oluşturulmalıdır.

Kuyruk setlerine `QueueSetHandle_t` türündeki değişkenler olan tanıtıcılar (handles) ile başvurulur. `xQueueCreateSet()` API işlevi bir kuyruk seti oluşturur ve oluşturulan kuyruk setine referans veren bir `QueueSetHandle_t` döndürür.

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength );
```

Liste 5.21 `xQueueCreateSet()` API işlev prototipi

`xQueueCreateSet()` parametreleri ve dönüş değeri:

- **`uxEventQueueLength`:** Bir kuyruk setinin üyesi olan bir kuyruk veri aldığında, alıcı kuyruğun tanıtıcısı kuyruk setine gönderilir. `uxEventQueueLength`, oluşturulmakta olan kuyruk setinin herhangi bir zamanda tutabileceği maksimum kuyruk tanıtıcısı sayısını tanımlar.

  Kuyruk tanıtıcıları, yalnızca set içindeki bir kuyruk veri aldığında bir kuyruk setine gönderilir. Bir kuyruk doluysa veri alamaz, bu nedenle setteki tüm kuyruklar doluysa kuyruk setine hiçbir kuyruk tanıtıcısı gönderilemez. Bu nedenle, kuyruk setinin bir defada tutması gereken maksimum öğe sayısı, setteki her bir kuyruğun uzunluklarının toplamıdır.

  Örnek olarak, sette üç boş kuyruk varsa ve her kuyruğun uzunluğu beş ise, setteki tüm kuyruklar dolmadan önce setteki kuyruklar toplam on beş öğe (üç kuyruk çarpı her biri beş öğe) alabilir. Bu örnekte kuyruk setinin kendisine gönderilen her öğeyi alabilmesini garanti etmek için `uxEventQueueLength` değeri on beş olarak ayarlanmalıdır.
- **Dönüş değeri:** `NULL` döndürülürse, kuyruk setini oluşturmak için yeterli bellek (heap memory) tahsis edilememiştir. Aksi takdirde, geçerli bir `QueueSetHandle_t` tanıtıcısı döndürülür.

5.6.3 xQueueAddToSet() API İşlevi

`xQueueAddToSet()`, bir kuyruğu veya semaforu bir kuyruk setine ekler.

```
BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore,

                           QueueSetHandle_t xQueueSet );
```

Liste 5.22 `xQueueAddToSet()` API işlev prototipi

5.6.4 xQueueSelectFromSet() API İşlevi

`xQueueSelectFromSet()` bir kuyruk setinden (queue set) okuma yapar. Set içindeki bir veya daha fazla kuyruk veri içerdiğinde, ilgili kuyruğun (veya semaforun) tanıtıcısını döndürür. Bu tanıtıcı kullanılarak, verinin aslında hangi kuyrukta olduğu belirlenip veri kuyruktan doğrudan okunabilir.

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet,

                                            const TickType_t xTicksToWait );
```

Liste 5.23 `xQueueSelectFromSet()` API işlev prototipi

5.7 Posta Kutusu (Mailbox) Oluşturmak İçin Kuyruk Kullanımı

Gömülü topluluğu içinde terminoloji konusunda bir fikir birliği yoktur ve 'posta kutusu' (mailbox) farklı RTOS'larda farklı anlamlara gelecektir. Bu kitapta, posta kutusu terimi uzunluğu bir olan bir kuyruğa atıfta bulunmak için kullanılmaktadır. Bir kuyruk, uygulamada kullanılma şekli nedeniyle posta kutusu olarak tanımlanabilir, bir kuyrukla işlevsel bir farkı olduğu için değil:

• Bir kuyruk, bir görevden diğerine veya bir kesme hizmet rutininden bir göreve veri göndermek için kullanılır. Gönderici kuyruğa bir öğe yerleştirir ve alıcı öğeyi kuyruktan çıkarır. Veri kuyruk aracılığıyla göndericiden alıcıya geçer.

• Posta kutusu, herhangi bir görev veya herhangi bir kesme hizmet rutini tarafından okunabilen verileri tutmak için kullanılır. Veri posta kutusundan geçmez, bunun yerine üzerine yazılana kadar posta kutusunda kalır. Gönderici posta kutusundaki değerin üzerine yazar. Alıcı posta kutusundaki değeri okur, ancak değeri posta kutusundan kaldırmaz.

Bu bölüm, bir kuyruğun posta kutusu olarak kullanılmasını sağlayan iki kuyruk API işlevini açıklamaktadır.

Liste 5.28, bir kuyruğun posta kutusu olarak kullanılmak üzere nasıl oluşturulduğunu göstermektedir.

```
/* Posta kutusu sabit boyutlu bir veri öğesi tutabilir. Veri öğesinin boyutu
   posta kutusu (kuyruk) oluşturulduğunda ayarlanır. Bu örnekte posta kutusu
   bir Example_t yapısı tutmak üzere oluşturulmuştur. Example_t, posta
   kutusunun en son ne zaman güncellendiğini not etmek için bir zaman damgası
   içerir. Bu örnekte kullanılan zaman damgası yalnızca gösterim amaçlıdır
   – bir posta kutusu uygulama yazarının istediği herhangi bir veriyi
   tutabilir ve verinin bir zaman damgası içermesi gerekmez. */
typedef struct xExampleStructure
{
    TickType_t xTimeStamp;
    uint32_t ulValue;
```

```
} Example_t;

/* Posta kutusu bir kuyruktur, dolayısıyla tanıtıcısı (handle) QueueHandle_t
   türünde bir değişkende saklanır. */
QueueHandle_t xMailbox;

void vAFunction( void )
{
    /* Posta kutusu olarak kullanılacak kuyruğu oluşturun. Kuyruk,
       aşağıda açıklanan xQueueOverwrite() API işlevi ile kullanılabilmesi
       için 1 uzunluğundadır. */
    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );
}
```

Liste 5.28 *Posta kutusu olarak kullanılmak üzere oluşturulan bir kuyruk*

5.7.1 xQueueOverwrite() API İşlevi

xQueueSendToBack() API işlevi gibi, xQueueOverwrite() API işlevi de bir kuyruğa veri gönderir. xQueueSendToBack()'ten farklı olarak, kuyruk zaten doluysa, xQueueOverwrite() kuyrukta zaten bulunan verinin üzerine yazar.

xQueueOverwrite() yalnızca uzunluğu bir olan kuyruklarla kullanılmalıdır. Üzerine yazma (overwrite) modu her zaman kuyruğun başına (front) yazar ve kuyruk başı işaretçisini günceller, ancak bekleyen mesajları güncellemez. Eğer configASSERT tanımlanmışsa, kuyruğun uzunluğu > 1 ise bir iddia (assert) tetiklenir.

Not: *Bir kesme hizmet rutininden asla* xQueueOverwrite() *çağırmayın. Bunun yerine kesme güvenli sürümü olan* xQueueOverwriteFromISR() *kullanılmalıdır.*

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );
```

Liste 5.29 *xQueueOverwrite() API işlev prototipi*

xQueueOverwrite() parametreleri ve dönüş değeri

- xQueue

  Verinin gönderildiği (yazıldığı) kuyruğun tanıtıcısı (handle). Kuyruk tanıtıcısı, kuyruğu oluşturmak için kullanılan xQueueCreate() veya xQueueCreateStatic() çağrısından döndürülmüş olacaktır.

- pvItemToQueue

  Kuyruğa kopyalanacak veriye yönelik bir işaretçi.

Kuyruğun tutabileceği her bir öğenin boyutu kuyruk oluşturulduğunda ayarlanır, bu nedenle bu kadar bayt pvItemToQueue'dan kuyruk depolama alanına kopyalanacaktır.

• **Dönüş değeri**

xQueueOverwrite() kuyruk dolu olduğunda bile kuyruğa yazar, bu nedenle pdPASS tek olası dönüş değeridir.

Liste 5.30, xQueueOverwrite()'ın Liste 5.28'de oluşturulan posta kutusuna (kuyruğa) yazmak için nasıl kullanıldığını göstermektedir.

```
void vUpdateMailbox( uint32_t ulNewValue )
{
    /* Example_t, Liste 5.28'de tanımlanmıştır. */
    Example_t xData;

    /* Yeni veriyi Example_t yapısına yazın. */
    xData.ulValue = ulNewValue;

    /* Example_t yapısında saklanan zaman damgası olarak
       RTOS tick sayısını kullanın. */
    xData.xTimeStamp = xTaskGetTickCount();

    /* Yapıyı posta kutusuna gönderin — posta kutusunda
       zaten bulunan verinin üzerine yazın. */
    xQueueOverwrite( xMailbox, &xData );
}
```

Liste 5.30 *xQueueOverwrite() API işlevinin kullanımı*

5.7.2 xQueuePeek() API İşlevi

`xQueuePeek()`bir kuyruktan öğeyi kuyruktan **kaldırmadan** alır (okur). xQueuePeek() kuyrukta saklanan veriyi değiştirmeden veya verilerin kuyrukta saklanma sırasını değiştirmeden kuyruğun başından (head) veri alır.

Not: *Bir kesme hizmet rutininden asla* xQueuePeek() *çağırmayın. Bunun yerine kesme güvenli sürümü olan* xQueuePeekFromISR() *kullanılmalıdır.*

xQueuePeek() işlevi xQueueReceive() ile aynı işlev parametrelerine ve dönüş değerine sahiptir.

```
BaseType_t xQueuePeek( QueueHandle_t xQueue,

                       void * const pvBuffer,

                       TickType_t xTicksToWait );
```

Liste 5.31 *xQueuePeek() API işlev prototipi*

```
BaseType_t vReadMailbox( Example_t *pxData )
{
    TickType_t xPreviousTimeStamp;
    BaseType_t xDataUpdated;

    /* Bu işlev bir Example_t yapısını posta kutusundan alınan en son
       değerle günceller. Yeni veri tarafından üzerine yazılmadan önce
       *pxData'da zaten bulunan zaman damgasını kaydedin. */
    xPreviousTimeStamp = pxData->xTimeStamp;

    /* pxData tarafından işaret edilen Example_t yapısını posta kutusunda
       bulunan verilerle güncelleyin. Burada xQueueReceive() kullanılsaydı
       posta kutusu boş kalırdı ve veri diğer görevler tarafından
       okunamazdı. xQueueReceive() yerine xQueuePeek() kullanmak
       verinin posta kutusunda kalmasını sağlar.

       Bir engelleme süresi belirtilmiştir, bu nedenle posta kutusunun
       boş olması durumunda çağıran görev Engellenmiş (Blocked) durumuna
       yerleştirilecektir. Sonsuz bir engelleme süresi kullanıldığından
       xQueuePeek()'den dönen değeri kontrol etmek gerekli değildir,
       çünkü xQueuePeek() yalnızca veri mevcut olduğunda dönecektir. */
    xQueuePeek( xMailbox, pxData, portMAX_DELAY );

    /* Posta kutusundan okunan değer bu işlevin en son çağrılmasından
       beri güncellenmişse pdTRUE döndürün. Aksi takdirde pdFALSE döndürün. */
```

```
    if( pxData->xTimeStamp > xPreviousTimeStamp )
    {
        xDataUpdated = pdTRUE;
    }
    else
    {
        xDataUpdated = pdFALSE;
    }

    return xDataUpdated;
}
```

Liste 5.32, xQueuePeek()'ın Liste 5.30'da gönderilen öğeyi posta kutusundan (kuyruktan) almak için nasıl kullanıldığını göstermektedir

6 Yazılım Zamanlayıcı (Software Timer) Yönetimi

6.1 Giriş

Yazılım zamanlayıcıları (software timer), bir işlevin gelecekte belirli bir zamanda veya sabit bir frekansla periyodik olarak yürütülmesini zamanlamak için kullanılır. Yazılım zamanlayıcısı tarafından yürütülen işleve yazılım zamanlayıcısının geri çağırma (callback) işlevi denir.

Yazılım zamanlayıcıları FreeRTOS çekirdeği tarafından uygulanır ve kontrol edilir. Donanım desteği gerektirmezler ve donanım zamanlayıcıları veya donanım sayaçları ile ilgili değildirler.

FreeRTOS'un maksimum verimliliği sağlamak için yenilikçi tasarım kullanma felsefesi doğrultusunda, bir yazılım zamanlayıcısının geri çağırma işlevi gerçekten yürütülmediği sürece yazılım zamanlayıcılarının herhangi bir işlem süresi kullanmadığını unutmayın.

Yazılım zamanlayıcı işlevselliği isteğe bağlıdır. Yazılım zamanlayıcı işlevselliğini dahil etmek için:

1. FreeRTOS kaynak dosyası FreeRTOS/Source/timers.c'yi projenizin bir parçası olarak derleyin.

2. Aşağıda ayrıntıları verilen sabitleri uygulamanın FreeRTOSConfig.h başlık dosyasında tanımlayın:
• `configUSE_TIMERS`
   FreeRTOSConfig.h'da `configUSE_TIMERS`'ı 1 olarak ayarlayın.
• `configTIMER_TASK_PRIORITY`
   Zamanlayıcı hizmet görevinin önceliğini 0 ile (`configMAX_PRIORITIES` - 1) arasında ayarlar.
• `configTIMER_QUEUE_LENGTH`
   Zamanlayıcı komut kuyruğunun herhangi bir anda tutabileceği maksimum işlenmemiş komut sayısını ayarlar.
• `configTIMER_TASK_STACK_DEPTH`
   Zamanlayıcı hizmet görevine ayrılan yığının (stack) boyutunu (bayt değil, kelime cinsinden) ayarlar.

6.1.1 Kapsam

Bu bölüm şunları kapsamaktadır:

- Yazılım zamanlayıcısı nedir?
- Geri çağırma (callback) işlevi nasıl yazılır?

- Tek atışlık (one-shot) ve otomatik yeniden yüklemeli (auto-reload) zamanlayıcılar arasındaki farklar.
- Zamanlayıcı nasıl oluşturulur, başlatılır, sıfırlanır (reset) ve süresi nasıl değiştirilir?

6.2 Yazılım Zamanlayıcısı Kanca (Callback) İşlevleri

Yazılım zamanlayıcı geri çağırma işlevleri C işlevleri olarak uygulanır. Bunlardaki tek özel şey prototipleridir; void döndürmeleri ve tek parametreleri olarak bir yazılım zamanlayıcısına tanıtıcı (handle) almaları gerekir. Geri çağırma işlevi prototipi Liste 6.1 ile gösterilmektedir.

```
void ATimerCallback( TimerHandle_t xTimer );
```

Liste 6.1 *Yazılım zamanlayıcı geri çağırma işlevi prototipi*

Yazılım zamanlayıcı geri çağırma işlevleri baştan sona yürütülür ve normal şekilde çıkar. Kısa tutulmalı ve Engellenmiş (Blocked) durumuna girmemelidirler.

Not: *Görüleceği gibi, yazılım zamanlayıcı geri çağırma işlevleri FreeRTOS çizgeleyicisi (scheduler) başlatıldığında otomatik olarak oluşturulan bir görevin bağlamında yürütülür. Bu nedenle, yazılım zamanlayıcı geri çağırma işlevlerinin çağıran görevi Engellenmiş durumuna sokacak FreeRTOS API işlevlerini asla çağırmaması çok önemlidir.* `xQueueReceive()` *gibi işlevleri çağırmak uygundur, ancak yalnızca işlevin* `xTicksToWait` *parametresi (işlevin engelleme süresini belirleyen) 0 olarak ayarlanmışsa.* `vTaskDelay()` *gibi işlevleri çağırmak uygun değildir, çünkü* `vTaskDelay()` *çağrıldığında çağıran görev her zaman Engellenmiş durumuna yerleştirilecektir.*

.

6.3 Zamanlayıcı Servis Görevi (Timer Service Task)

6.3.1 Bir Yazılım Zamanlayıcısının Periyodu

Bir yazılım zamanlayıcısının 'periyodu', yazılım zamanlayıcısının başlatılması ile yazılım zamanlayıcısının geri çağırma işlevinin yürütülmesi arasındaki süredir.

6.3.2 Tek Seferlik (One-shot) ve Otomatik Yeniden Yükleme (Auto-reload) Zamanlayıcıları

İki tür yazılım zamanlayıcısı vardır:
1. Tek seferlik (one-shot) zamanlayıcılar

Başlatıldığında, tek seferlik bir zamanlayıcı geri çağırma işlevini yalnızca bir kez yürütür. Tek seferlik bir zamanlayıcı manuel olarak yeniden başlatılabilir, ancak kendini yeniden başlatmaz.

2. Otomatik yeniden yükleme (auto-reload) zamanlayıcıları

Başlatıldığında, otomatik yeniden yükleme zamanlayıcısı her süresi dolduğunda kendini yeniden başlatır ve geri çağırma işlevinin periyodik olarak yürütülmesini sağlar.

Şekil 6.1, tek seferlik bir zamanlayıcı ile otomatik yeniden yükleme zamanlayıcısı arasındaki davranış farkını göstermektedir. Kesikli dikey çizgiler bir tick kesmesinin oluştuğu zamanları işaretler.

Mastering the FreeRTOS Real Time Kernel

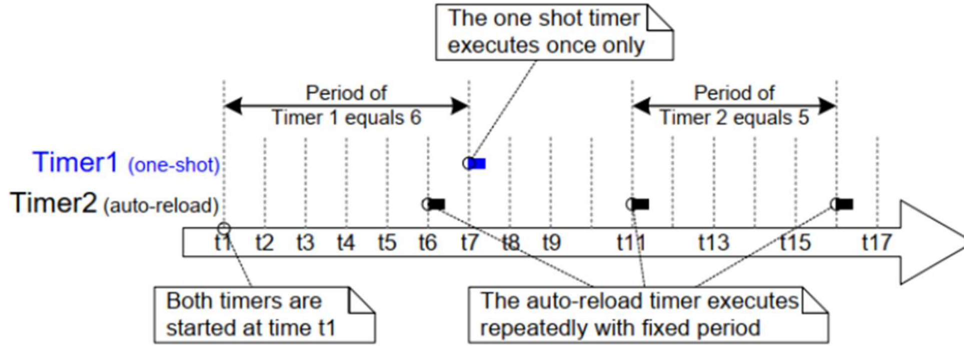


Figure 6.1 The difference in behavior between one-shot and auto-reload software timers

Şekil 6.1 *Tek seferlik ve otomatik yeniden yükleme yazılım zamanlayıcıları arasındaki davranış farkı*

6.3.3 Yazılım Zamanlayıcı Durumları

Bir yazılım zamanlayıcısı aşağıdaki iki durumdan birinde olabilir:

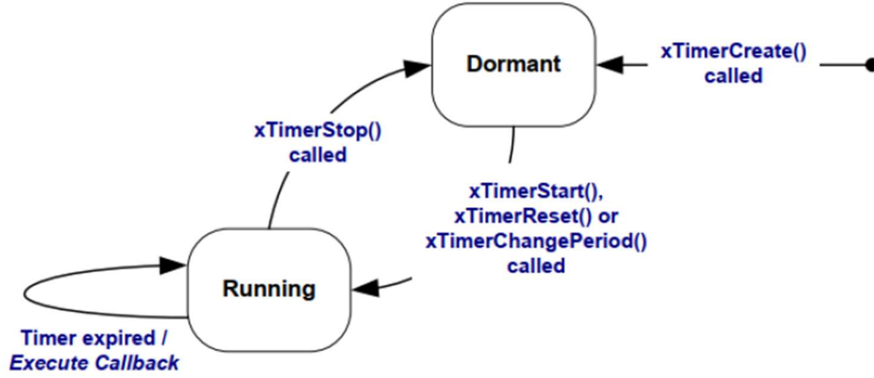
• Uyku (Dormant)

Uyku durumundaki bir yazılım zamanlayıcısı mevcuttur ve tanıtıcısı (handle) ile referans gösterilebilir, ancak çalışmıyordur, dolayısıyla geri çağırma işlevleri yürütülmez.

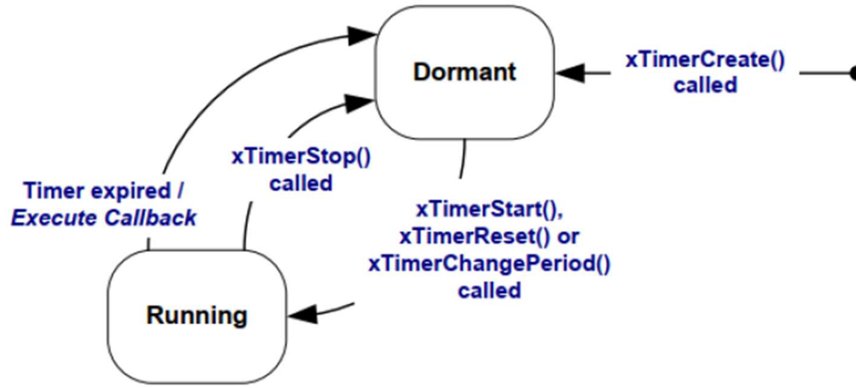
• Çalışıyor (Running)

Çalışıyor durumundaki bir yazılım zamanlayıcısı, zamanlayıcının Çalışıyor durumuna girmesinden veya zamanlayıcının en son sıfırlanmasından bu yana periyoduna eşit bir süre geçtikten sonra geri çağırma işlevini yürütecektir.

Şekil 6.2 ve Şekil 6.3, sırasıyla otomatik yeniden yükleme zamanlayıcısı ve tek seferlik zamanlayıcı için Uyku ve Çalışıyor durumları arasındaki olası geçişleri göstermektedir. İki diyagram arasındaki temel fark, zamanlayıcının süresi dolduktan sonra girilen durumdur; otomatik yeniden yükleme zamanlayıcısı geri çağırma işlevini yürütür ve ardından Çalışıyor durumuna yeniden girer, tek seferlik zamanlayıcı ise geri çağırma işlevini yürütür ve ardından Uyku durumuna girer.



***Şekil 6.2** Otomatik yeniden yükleme yazılım zamanlayıcısı durumları ve geçişleri*



***Şekil 6.3** Tek seferlik yazılım zamanlayıcısı durumları ve geçişleri*

`xTimerDelete()` API işlevi bir zamanlayıcıyı siler. Bir zamanlayıcı herhangi bir zamanda silinebilir. İşlev prototipi Liste 6.2 ile gösterilmektedir.

```
BaseType_t xTimerDelete( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

***Liste 6.2** xTimerDelete() API işlev prototipi*

xTimerDelete() parametreleri ve dönüş değeri

- `xTimer`
  Silinen zamanlayıcının tanıtıcısı (handle).
- `xTicksToWait`
  Silme komutunun zamanlayıcı komut kuyruğuna başarıyla gönderilmesi için çağıran görevin Engellenmiş durumunda bekletilmesi gereken süreyi tick cinsinden belirtir. `xTimerDelete()` çağrıldığında kuyruk zaten dolu olması durumunda geçerlidir. `xTicksToWait`, çizgeleyici (scheduler) başlatılmadan önce `xTimerDelete()` çağrılırsa yoksayılır.
- Dönüş değeri

İki olası dönüş değeri vardır:

- `pdPASS`

  Komut zamanlayıcı komut kuyruğuna başarıyla gönderildiğinde `pdPASS` döndürülür.
- `pdFAIL`

  `xBlockTime` tick geçtikten sonra bile silme komutu zamanlayıcı komut kuyruğuna gönderilemezse `pdFAIL` döndürülür.

6.4 Zamanlayıcı Türleri

6.4.1 RTOS Arka Plan (Daemon) (Zamanlayıcı Hizmet) Görevi

Tüm yazılım zamanlayıcı geri çağırma işlevleri aynı RTOS arka plan (daemon) (veya 'zamanlayıcı hizmet') görevinin bağlamında yürütülür.

[Not 10]: Görev eskiden 'zamanlayıcı hizmet görevi' olarak adlandırılıyordu, çünkü başlangıçta yalnızca yazılım zamanlayıcı geri çağırma işlevlerini yürütmek için kullanılıyordu. Artık aynı görev başka amaçlar için de kullanılmaktadır, bu nedenle daha genel olan 'RTOS arka plan görevi' adıyla bilinmektedir.

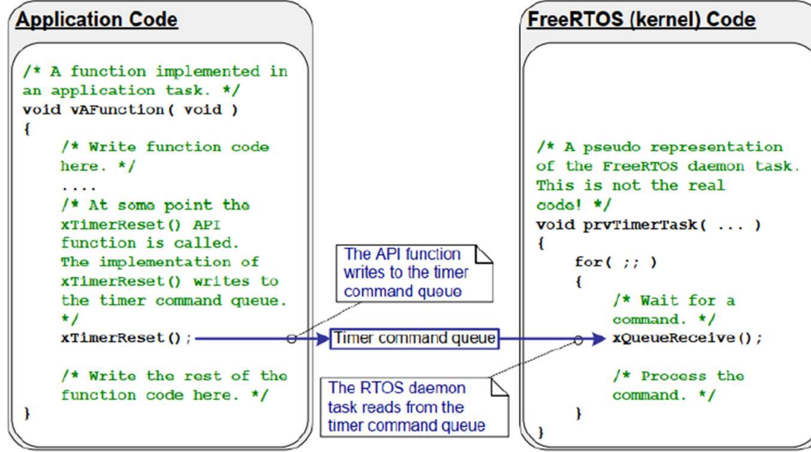
Arka plan görevi, çizgeleyici (scheduler) başlatıldığında otomatik olarak oluşturulan standart bir FreeRTOS görevidir. Önceliği ve yığın boyutu sırasıyla configTIMER_TASK_PRIORITY ve configTIMER_TASK_STACK_DEPTH derleme zamanı yapılandırma sabitleri tarafından ayarlanır. Her iki sabit da FreeRTOSConfig.h içinde tanımlanır.

Yazılım zamanlayıcı geri çağırma işlevleri, çağıran görevi Engellenmiş durumuna sokacak FreeRTOS API işlevlerini çağırmamalıdır, aksi takdirde arka plan görevi Engellenmiş durumuna girecektir.

6.4.2 Otomatik Yeniden Yüklemeli (Auto-reload) Zamanlayıcılar

Yazılım zamanlayıcı API işlevleri, çağıran görevden arka plan görevine 'zamanlayıcı komut kuyruğu' adı verilen bir kuyruk üzerinden komutlar gönderir. Bu Şekil 6.4'te gösterilmektedir. Komut örnekleri arasında 'bir zamanlayıcı başlat', 'bir zamanlayıcıyı durdur' ve 'bir zamanlayıcıyı sıfırla' yer alır.

Zamanlayıcı komut kuyruğu, çizgeleyici (scheduler) başlatıldığında otomatik olarak oluşturulan standart bir FreeRTOS kuyruğudur. Zamanlayıcı komut kuyruğunun uzunluğu FreeRTOSConfig.h'daki configTIMER_QUEUE_LENGTH derleme zamanı yapılandırma sabiti tarafından ayarlanır.



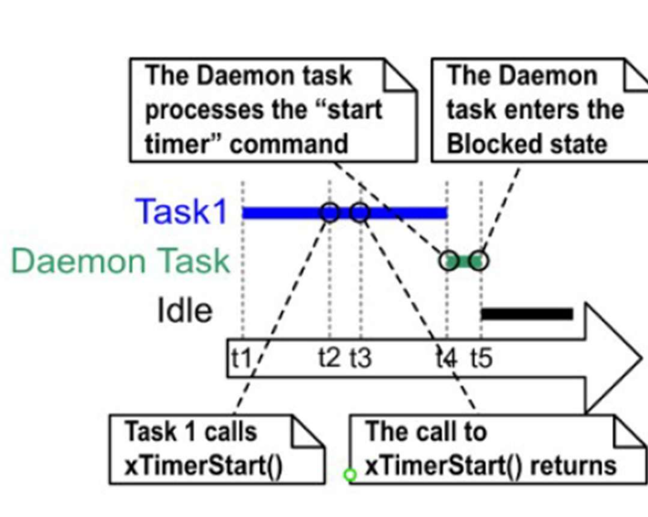
Şekil 6.4 *Bir yazılım zamanlayıcı API işlevi tarafından RTOS arka plan görevi ile iletişim kurmak için kullanılan zamanlayıcı komut kuyruğu*

6.4.3 Arka Plan Görevi Çizgeleme

Arka plan görevi diğer tüm FreeRTOS görevleri gibi çizgelenir; yalnızca çalışabilecek en yüksek öncelikli görev olduğunda komutları işler veya zamanlayıcı geri çağırma işlevlerini yürütür.

Şekil 6.5 ve Şekil 6.6, configTIMER_TASK_PRIORITY ayarının yürütme düzenini nasıl etkilediğini göstermektedir.

Şekil 6.5, arka plan görevinin önceliğinin `xTimerStart()` API işlevini çağıran bir görevin önceliğinin altında olduğundaki yürütme düzenini göstermektedir.



Şekil 6.5 *xTimerStart() çağıran görevin önceliği arka plan görevinin önceliğinin üstünde olduğundaki yürütme düzeni*

Şekil 6.5'e atıfla, burada Görev 1'in önceliği arka plan görevinin önceliğinden yüksektir ve arka plan görevinin önceliği Boşta görevinin önceliğinden yüksektir:

1. t1 zamanında
   Görev 1 Çalışıyor durumundadır ve arka plan görevi Engellenmiş durumdadır.
   Arka plan görevi, zamanlayıcı komut kuyruğuna bir komut gönderilirse (bu durumda komutu işler) veya bir yazılım zamanlayıcısının süresi dolarsa (bu durumda yazılım zamanlayıcısının geri çağırma işlevini yürütür) Engellenmiş durumdan çıkacaktır.

2. t2 zamanında
   Görev 1 `xTimerStart()`'ı çağırır.
`xTimerStart()` zamanlayıcı komut kuyruğuna bir komut göndererek arka plan görevinin Engellenmiş durumdan çıkmasına neden olur. Görev 1'in önceliği arka plan görevinin önceliğinden yüksek olduğundan, arka plan görevi Görev 1'i öncelemez (pre-empt etmez).
   Görev 1 hâlâ Çalışıyor durumundadır ve arka plan görevi Engellenmiş durumdan çıkıp Hazır durumuna girmiştir.

3. t3 zamanında
   Görev 1 `xTimerStart()` API işlevini yürütmeyi tamamlar. Görev 1, işlevin başından sonuna kadar Çalışıyor durumundan ayrılmadan `xTimerStart()`'ı yürütmüştür.

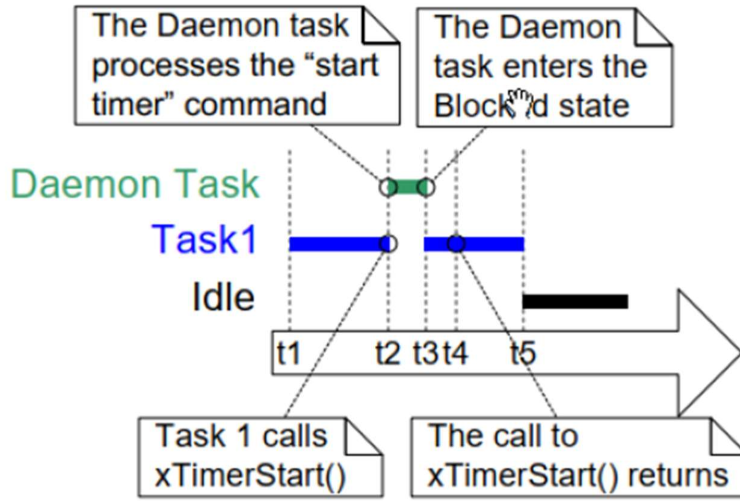
4. t4 zamanında
   Görev 1, kendisinin Engellenmiş durumuna girmesine neden olan bir API işlevi çağırır. Arka plan görevi artık Hazır durumundaki en yüksek öncelikli görevdir, bu nedenle çizgeleyici arka plan görevini Çalışıyor durumuna girecek görev olarak seçer. Arka plan görevi daha sonra Görev 1 tarafından zamanlayıcı komut kuyruğuna gönderilen komutu işlemeye başlar.

***Not:** Başlatılan yazılım zamanlayıcısının süresinin dolacağı zaman, 'zamanlayıcıyı başlat' komutunun zamanlayıcı komut kuyruğuna gönderildiği zamandan itibaren hesaplanır — arka plan görevinin zamanlayıcı komut kuyruğundan 'zamanlayıcıyı başlat' komutunu aldığı zamandan itibaren değil.*

5. t5 zamanında
   Arka plan görevi, Görev 1 tarafından kendisine gönderilen komutu işlemeyi tamamlamıştır ve zamanlayıcı komut kuyruğundan daha fazla veri almaya çalışır. Zamanlayıcı komut kuyruğu boştur, bu nedenle arka plan görevi Engellenmiş durumuna yeniden girer.

   Boşta görevi artık Hazır durumundaki en yüksek öncelikli görevdir, bu nedenle çizgeleyici Boşta görevini Çalışıyor durumuna girecek görev olarak seçer.

Şekil 6.6, Şekil 6.5'te gösterilene benzer bir senaryo göstermektedir, ancak bu kez arka plan görevinin önceliği xTimerStart() çağıran görevin önceliğinin üstündedir.



Şekil 6.6 *xTimerStart() çağıran görevin önceliği arka plan görevinin önceliğinin altında olduğundaki yürütme düzeni*

Şekil 6.6'ya atıfla, burada arka plan görevinin önceliği Görev 1'in önceliğinden yüksektir ve Görev 1'in önceliği Boşta görevinin önceliğinden yüksektir:

1. t1 zamanında

   Daha önce olduğu gibi, Görev 1 Çalışıyor durumundadır ve arka plan görevi Engellenmiş durumdadır.

2. t2 zamanında

   Görev 1 `xTimerStart()`'ı çağırır.
`xTimerStart()` zamanlayıcı komut kuyruğuna bir komut göndererek arka plan görevinin Engellenmiş durumdan çıkmasına neden olur. Arka plan görevinin önceliği Görev 1'in önceliğinden yüksek olduğundan, çizgeleyici arka plan görevini Çalışıyor durumuna girecek görev olarak seçer.

   Görev 1, `xTimerStart()` işlevini yürütmeyi tamamlamadan önce arka plan görevi tarafından öncelenmiştir ve artık Hazır durumundadır.

   Arka plan görevi, Görev 1 tarafından zamanlayıcı komut kuyruğuna gönderilen komutu işlemeye başlar.

3. t3 zamanında

   Arka plan görevi, Görev 1 tarafından kendisine gönderilen komutu işlemeyi tamamlamıştır ve zamanlayıcı komut kuyruğundan daha fazla veri almaya çalışır. Zamanlayıcı komut kuyruğu boştur, bu nedenle arka plan görevi Engellenmiş durumuna yeniden girer.

   Görev 1 artık Hazır durumundaki en yüksek öncelikli görevdir, bu nedenle çizgeleyici Görev 1'i Çalışıyor durumuna girecek görev olarak seçer.

4. t4 zamanında

   Görev 1, `xTimerStart()` işlevini yürütmeyi tamamlamadan önce arka plan görevi tarafından öncelenmişti ve `xTimerStart()`'tan yalnızca Çalışıyor durumuna yeniden girdikten sonra çıkar (geri döner).

5. t5 zamanında
Görev 1, kendisinin Engellenmiş durumuna girmesine neden olan bir API işlevi çağırır. Boşta görevi artık Hazır durumundaki en yüksek öncelikli görevdir, bu nedenle çizgeleyici Boşta görevini Çalışıyor durumuna girecek görev olarak seçer. Şekil 6.5'te gösterilen senaryoda, Görev 1'in zamanlayıcı komut kuyruğuna komut göndermesi ile arka plan görevinin komutu alması ve işlemesi arasında zaman geçmiştir. Şekil 6.6'da gösterilen senaryoda ise, arka plan görevi, komutu gönderen Görev 1 işlevden dönmeden önce Görev 1 tarafından kendisine gönderilen komutu almış ve işlemiştir.

Zamanlayıcı komut kuyruğuna gönderilen komutlar bir zaman damgası içerir. Bu zaman damgası, bir uygulama görevi tarafından gönderilen bir komut ile aynı komutun arka plan görevi tarafından işlenmesi arasında geçen süreyi hesaba katmak için kullanılır. Örneğin, periyodu 10 tick olan bir zamanlayıcıyı başlatmak için bir 'zamanlayıcıyı başlat' komutu gönderilirse, zaman damgası başlatılan zamanlayıcının komutun gönderildiği andan 10 tick sonra süresinin dolmasını sağlamak için kullanılır — komutun arka plan görevi tarafından işlendiği andan 10 tick sonra değil.

6.5 Yazılım Zamanlayıcılarını Kullanma (Using Software Timers)

6.5.1 xTimerCreate() ve xTimerCreateStatic() API İşlevleri

FreeRTOS ayrıca bir zamanlayıcıyı derleme zamanında statik olarak oluşturmak için gereken belleği ayıran `xTimerCreateStatic()` işlevini de içerir: Bir yazılım zamanlayıcısı kullanılmadan önce açıkça oluşturulmalıdır.

Yazılım zamanlayıcıları TimerHandle_t türünde değişkenler ile referans gösterilir. xTimerCreate() bir yazılım zamanlayıcısı oluşturmak için kullanılır ve oluşturduğu yazılım zamanlayıcısına referans vermek için bir TimerHandle_t döndürür. Yazılım zamanlayıcıları Uyku (Dormant) durumunda oluşturulur.

Yazılım zamanlayıcıları çizgeleyici çalışmadan önce veya çizgeleyici başlatıldıktan sonra bir görevden oluşturulabilir.

Bölüm 2.5: Veri Türleri ve Kodlama Stili Rehberi kullanılan veri türlerini ve adlandırma kurallarını açıklamaktadır.

```
TimerHandle_t xTimerCreate( const char * const pcTimerName,
                            const TickType_t xTimerPeriodInTicks,
                            const BaseType_t xAutoReload,
                            void * const pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction );
```

Liste 6.3 *xTimerCreate() API işlev prototipi*

xTimerCreate() parametreleri ve dönüş değeri

- `pcTimerName`
  Zamanlayıcı için açıklayıcı bir ad. FreeRTOS tarafından hiçbir şekilde kullanılmaz. Yalnızca hata ayıklama yardımcısı olarak dahil edilmiştir. Bir zamanlayıcıyı insan

tarafından okunabilir bir adla tanımlamak, tanıtıcısı (handle) ile tanımlamaya çalışmaktan çok daha basittir.

- `xTimerPeriodInTicks`
  Zamanlayıcının tick cinsinden belirtilen periyodu. `pdMS_TO_TICKS()` makrosu milisaniye cinsinden belirtilen bir süreyi tick cinsinden belirtilen bir süreye dönüştürmek için kullanılabilir. 0 olamaz.

- `xAutoReload`
  Otomatik yeniden yükleme zamanlayıcısı oluşturmak için `xAutoReload`'u `pdTRUE` olarak ayarlayın. Tek seferlik zamanlayıcı oluşturmak için `xAutoReload`'u `pdFALSE` olarak ayarlayın.

- `pvTimerID`
  Her yazılım zamanlayıcısının bir ID değeri vardır. ID bir void işaretçisidir ve uygulama yazarı tarafından herhangi bir amaç için kullanılabilir. ID, özellikle aynı geri çağırma işlevi birden fazla yazılım zamanlayıcısı tarafından kullanıldığında yararlıdır, çünkü zamanlayıcıya özgü depolama sağlamak için kullanılabilir. Bu bölümde bir zamanlayıcı ID'sinin kullanımı bir örnekle gösterilmektedir. `pvTimerID` oluşturulan görevin ID'si için başlangıç değerini ayarlar.

- `pxCallbackFunction`
  Yazılım zamanlayıcı geri çağırma işlevleri, Liste 6.1'de gösterilen prototipe uygun C işlevleridir. `pxCallbackFunction` parametresi, oluşturulan yazılım zamanlayıcısının geri çağırma işlevi olarak kullanılacak işleve yönelik bir işaretçidir (aslında yalnızca işlev adıdır).

- Dönüş değeri
  NULL döndürülürse, gerekli veri yapısını ayırmak için yeterli yığın bellek (heap memory) bulunmadığından yazılım zamanlayıcısı oluşturulamaz.
  NULL olmayan bir değer döndürülürse, yazılım zamanlayıcısının başarıyla oluşturulduğunu gösterir. Döndürülen değer, oluşturulan zamanlayıcının tanıtıcısıdır (handle).

  Bölüm 3, yığın bellek yönetimi hakkında daha fazla bilgi sağlamaktadır.

6.5.2 xTimerStart() API İşlevi

`xTimerStart()` Uyku (Dormant) durumundaki bir yazılım zamanlayıcısını başlatmak veya Çalışıyor durumundaki bir yazılım zamanlayıcısını sıfırlamak (yeniden başlatmak) için kullanılır. `xTimerStop()` Çalışıyor durumundaki bir yazılım zamanlayıcısını durdurmak için kullanılır. Bir yazılım zamanlayıcısını durdurmak, zamanlayıcıyı Uyku durumuna geçirmekle aynı anlama gelir.

`xTimerStart()` çizgeleyici başlatılmadan önce çağrılabilir, ancak bu yapıldığında, yazılım zamanlayıcısı çizgeleyicinin başladığı zamana kadar gerçekten başlamayacaktır.

Not: *Bir kesme hizmet rutininden asla xTimerStart() çağırmayın. Bunun yerine kesme güvenli sürümü olan xTimerStartFromISR() kullanılmalıdır.*

```
BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Liste 6.4 *xTimerStart() API işlev prototipi*

xTimerStart() parametreleri ve dönüş değeri

- `xTimer`

  Başlatılan veya sıfırlanan yazılım zamanlayıcısının tanıtıcısı (handle). Tanıtıcı, yazılım zamanlayıcısını oluşturmak için kullanılan `xTimerCreate()` çağrısından döndürülmüş olacaktır.

- `xTicksToWait`

  `xTimerStart()` arka plan görevine 'zamanlayıcıyı başlat' komutunu göndermek için zamanlayıcı komut kuyruğunu kullanır. `xTicksToWait` kuyruk zaten dolu olması durumunda çağıran görevin zamanlayıcı komut kuyruğunda yer açılması için Engellenmiş durumda kalması gereken maksimum süreyi belirtir. `xTimerStart()`, `xTicksToWait` sıfırsa ve zamanlayıcı komut kuyruğu zaten doluysa anında geri döner.

  Engelleme süresi tick periyotları cinsinden belirtilir, bu nedenle temsil ettiği mutlak süre tick frekansına bağlıdır. pdMS_TO_TICKS() makrosu milisaniye cinsinden belirtilen bir süreyi tick cinsinden belirtilen bir süreye dönüştürmek için kullanılabilir.

  FreeRTOSConfig.h'da `INCLUDE_vTaskSuspend` 1 olarak ayarlanmışsa, `xTicksToWait`'i `portMAX_DELAY` olarak ayarlamak, çağıran görevin zamanlayıcı komut kuyruğunda yer açılmasını süresiz olarak (zaman aşımı olmadan) Engellenmiş durumda beklemesiyle sonuçlanacaktır.

  Eğer `xTimerStart()` çizgeleyici başlatılmadan önce çağrılırsa, `xTicksToWait` değeri yoksayılır ve `xTimerStart()`, `xTicksToWait` sıfırmış gibi davranır.

- Dönüş değeri

  İki olası dönüş değeri vardır:

  - `pdPASS`

    Yalnızca 'zamanlayıcıyı başlat' komutu zamanlayıcı komut kuyruğuna başarıyla gönderildiğinde döndürülür.

  - `pdFAIL`

    Kuyruk zaten dolu olduğu için 'zamanlayıcıyı başlat' komutu zamanlayıcı komut kuyruğuna yazılamadığında döndürülür.

Örnek 6.1 *Tek seferlik ve otomatik yeniden yükleme zamanlayıcıları oluşturma*

Bu örnek, Liste 6.5'te gösterildiği gibi tek seferlik bir zamanlayıcı ve otomatik yeniden yükleme zamanlayıcısı oluşturur ve başlatır.

```
/* Tek seferlik ve otomatik yeniden yükleme zamanlayıcılarına atanan
   periyotlar sırasıyla 3,333 saniye ve yarım saniyedir. */
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
```

```
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;
    BaseType_t xTimer1Started, xTimer2Started;

    /* Tek seferlik zamanlayıcıyı oluşturun, tanıtıcıyı (handle)
       xOneShotTimer'da saklayın. */
    xOneShotTimer = xTimerCreate(
        /* Yazılım zamanlayıcısı için metin adı - FreeRTOS tarafından
kullanılmaz. */
                            "OneShot",
        /* Yazılım zamanlayıcısının tick cinsinden periyodu. */
                            mainONE_SHOT_TIMER_PERIOD,
        /* uxAutoRealod'u pdFALSE ayarlamak tek seferlik zamanlayıcı oluşturur.
*/
                            pdFALSE,
        /* Bu örnek zamanlayıcı id'sini kullanmaz. */
                            0,
        /* Oluşturulan yazılım zamanlayıcısı tarafından kullanılacak geri
çağırma işlevi. */
                            prvOneShotTimerCallback );

    /* Otomatik yeniden yükleme zamanlayıcısını oluşturun, tanıtıcıyı
       xAutoReloadTimer'da saklayın. */
    xAutoReloadTimer = xTimerCreate(
        /* Yazılım zamanlayıcısı için metin adı - FreeRTOS tarafından
kullanılmaz. */
                            "AutoReload",
        /* Yazılım zamanlayıcısının tick cinsinden periyodu. */
                            mainAUTO_RELOAD_TIMER_PERIOD,
        /* uxAutoRealod'u pdTRUE ayarlamak otomatik yeniden yükleme
zamanlayıcısı oluşturur. */
                            pdTRUE,
        /* Bu örnek zamanlayıcı id'sini kullanmaz. */
                            0,
        /* Oluşturulan yazılım zamanlayıcısı tarafından kullanılacak geri
çağırma işlevi. */
                            prvAutoReloadTimerCallback );

    /* Yazılım zamanlayıcılarının oluşturulup oluşturulmadığını kontrol edin.
*/
    if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
    {
        /* Yazılım zamanlayıcılarını 0 engelleme süresiyle başlatın (engelleme
yok).
           Çizgeleyici henüz başlatılmadığından burada belirtilen herhangi bir
           engelleme süresi yine de yoksayılacaktır. */
        xTimer1Started = xTimerStart( xOneShotTimer, 0 );
        xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

        /* xTimerStart() uygulaması zamanlayıcı komut kuyruğunu kullanır
           ve zamanlayıcı komut kuyruğu dolarsa xTimerStart() başarısız olur.
           Zamanlayıcı hizmet görevi çizgeleyici başlatılana kadar
oluşturulmaz,
```

```
            bu nedenle komut kuyruğuna gönderilen tüm komutlar çizgeleyici
            başlatılana kadar kuyrukta kalacaktır. Her iki xTimerStart()
            çağrısının da başarılı olup olmadığını kontrol edin. */
         if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
         {
             /* Çizgeleyiciyi başlatın. */
             vTaskStartScheduler();
         }
     }

     /* Her zamanki gibi, bu satıra ulaşılmamalıdır. */
     for( ;; );
}
```

```
/* Tek seferlik ve otomatik yeniden yükleme zamanlayıcılarına atanan
   periyotlar sırasıyla 3,333 saniye ve yarım saniyedir. */
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;
    BaseType_t xTimer1Started, xTimer2Started;

    /* Tek seferlik zamanlayıcıyı oluşturun, tanıtıcıyı (handle)
       xOneShotTimer'da saklayın. */
    xOneShotTimer = xTimerCreate(
        /* Yazılım zamanlayıcısı için metin adı - FreeRTOS tarafından
kullanılmaz. */
                            "OneShot",
        /* Yazılım zamanlayıcısının tick cinsinden periyodu. */
                            mainONE_SHOT_TIMER_PERIOD,
        /* uxAutoRealod'u pdFALSE ayarlamak tek seferlik zamanlayıcı oluşturur.
*/
                            pdFALSE,
        /* Bu örnek zamanlayıcı id'sini kullanmaz. */
                            0,
        /* Oluşturulan yazılım zamanlayıcısı tarafından kullanılacak geri
çağırma işlevi. */
                            prvOneShotTimerCallback );

    /* Otomatik yeniden yükleme zamanlayıcısını oluşturun, tanıtıcıyı
       xAutoReloadTimer'da saklayın. */
    xAutoReloadTimer = xTimerCreate(
        /* Yazılım zamanlayıcısı için metin adı - FreeRTOS tarafından
kullanılmaz. */
                            "AutoReload",
        /* Yazılım zamanlayıcısının tick cinsinden periyodu. */
                            mainAUTO_RELOAD_TIMER_PERIOD,
```

```
        /* uxAutoRealod'u pdTRUE ayarlamak otomatik yeniden yükleme
zamanlayıcısı oluşturur. */
                            pdTRUE,
        /* Bu örnek zamanlayıcı id'sini kullanmaz. */
                            0,
        /* Oluşturulan yazılım zamanlayıcısı tarafından kullanılacak geri
çağırma işlevi. */
                            prvAutoReloadTimerCallback );

    /* Yazılım zamanlayıcılarının oluşturulup oluşturulmadığını kontrol edin.
*/
    if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
    {
        /* Yazılım zamanlayıcılarını 0 engelleme süresiyle başlatın (engelleme
yok).
           Çizgeleyici henüz başlatılmadığından burada belirtilen herhangi bir
           engelleme süresi yine de yoksayılacaktır. */
        xTimer1Started = xTimerStart( xOneShotTimer, 0 );
        xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

        /* xTimerStart() uygulaması zamanlayıcı komut kuyruğunu kullanır
           ve zamanlayıcı komut kuyruğu dolarsa xTimerStart() başarısız olur.
           Zamanlayıcı hizmet görevi çizgeleyici başlatılana kadar
oluşturulmaz,
           bu nedenle komut kuyruğuna gönderilen tüm komutlar çizgeleyici
           başlatılana kadar kuyrukta kalacaktır. Her iki xTimerStart()
           çağrısının da başarılı olup olmadığını kontrol edin. */
        if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
        {
            /* Çizgeleyiciyi başlatın. */
            vTaskStartScheduler();
        }
    }

    /* Her zamanki gibi, bu satıra ulaşılmamalıdır. */
    for( ;; );
}
```

Liste 6.5 *Örnek 6.1'de kullanılan zamanlayıcıları oluşturma ve başlatma*

Zamanlayıcıların geri çağırma işlevleri her çağrıldıklarında bir mesaj yazdırırlar. Tek seferlik zamanlayıcı geri çağırma işlevinin uygulaması Liste 6.6'da gösterilmektedir. Otomatik yeniden yükleme zamanlayıcı geri çağırma işlevinin uygulaması Liste 6.7'de gösterilmektedir.

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    /* Geçerli tick sayısını alın. */
    xTimeNow = xTaskGetTickCount();
```

```
    /* Geri çağırmanın yürütüldüğü zamanı gösteren bir dize çıktılayın. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* Dosya kapsamı değişkeni. */
    ulCallCount++;
}
```

Liste 6.6 *Örnek 6.1'de tek seferlik zamanlayıcı tarafından kullanılan geri çağırma işlevi*

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    /* Geçerli tick sayısını alın. */
    xTimeNow = xTaskGetTickCount();

    /* Geri çağırmanın yürütüldüğü zamanı gösteren bir dize çıktılayın. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);

    ulCallCount++;
}
```

Liste 6.7 *Örnek 6.1'de otomatik yeniden yükleme zamanlayıcısı tarafından kullanılan geri çağırma işlevi*

Bu örneği çalıştırmak Şekil 6.7'de gösterilen çıktıyı üretir. Şekil 6.7, otomatik yeniden yükleme zamanlayıcısının geri çağırma işlevinin 500 tick'lik sabit bir periyotla yürütüldüğünü (Liste 6.5'te mainAUTO_RELOAD_TIMER_PERIOD 500 olarak ayarlanmıştır) ve tek seferlik zamanlayıcının geri çağırma işlevinin yalnızca bir kez, tick sayısı 3333 olduğunda yürütüldüğünü (Liste 6.5'te mainONE_SHOT_TIMER_PERIOD 3333 olarak ayarlanmıştır) göstermektedir.

```
C:\Windows\system32\cmd.exe - rtosdemo

C:\temp>rtosdemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
Auto-reload timer callback executing 3000
One-shot timer callback executing 3333
Auto-reload timer callback executing 3500
Auto-reload timer callback executing 4000
Auto-reload timer callback executing 4500
Auto-reload timer callback executing 5000
Auto-reload timer callback executing 5500
Auto-reload timer callback executing 6000
Auto-reload timer callback executing 6500
Auto-reload timer callback executing 7000
Auto-reload timer callback executing 7500
Auto-reload timer callback executing 8000
```

Şekil 6.7 *Örnek 6.1 yürütüldüğünde üretilen çıktı*

6.6 Zamanlayıcı Tanımlayıcısı (Timer ID)

Her yazılım zamanlayıcısının bir ID'si vardır; bu, uygulama yazarı tarafından herhangi bir amaç için kullanılabilen bir etiket değeridir. ID bir void işaretçisinde (void *) saklanır, bu nedenle doğrudan bir tamsayı değeri saklayabilir, başka herhangi bir nesneye işaret edebilir veya bir işlev işaretçisi olarak kullanılabilir.

Yazılım zamanlayıcısı oluşturulduğunda ID'ye bir başlangıç değeri atanır, ardından ID `vTimerSetTimerID()` API işlevi kullanılarak güncellenebilir ve `pvTimerGetTimerID()` API işlevi kullanılarak sorgulanabilir.
Diğer yazılım zamanlayıcı API işlevlerinden farklı olarak, `vTimerSetTimerID()` ve `pvTimerGetTimerID()` zamanlayıcı komut kuyruğuna bir komut göndermeden doğrudan yazılım zamanlayıcısına erişir.

6.6.1 vTimerSetTimerID() API İşlevi

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

Liste 6.8 *vTimerSetTimerID() API işlev prototipi*

vTimerSetTimerID() parametreleri

- `xTimer`

  Yeni bir ID değeriyle güncellenen yazılım zamanlayıcısının tanıtıcısı (handle). Tanıtıcı, yazılım zamanlayıcısını oluşturmak için kullanılan `xTimerCreate()` çağrısından döndürülmüş olacaktır.
- `pvNewID`

  Yazılım zamanlayıcısının ID'sinin ayarlanacağı değer.

6.6.2 pvTimerGetTimerID() API İşlevi

```
void *pvTimerGetTimerID( const TimerHandle_t xTimer );
```

Liste 6.9 *pvTimerGetTimerID() API işlev prototipi*

pvTimerGetTimerID() parametreleri ve dönüş değeri

- `xTimer`

  Sorgulanan yazılım zamanlayıcısının tanıtıcısı (handle). Tanıtıcı, yazılım zamanlayıcısını oluşturmak için kullanılan `xTimerCreate()` çağrısından döndürülmüş olacaktır.

- Dönüş değeri

  Sorgulanan yazılım zamanlayıcısının ID'si.

Örnek 6.2 *Geri çağırma işlevi parametresi ve yazılım zamanlayıcı ID'sinin kullanımı*

Aynı geri çağırma işlevi birden fazla yazılım zamanlayıcısına atanabilir. Bu yapıldığında, hangi yazılım zamanlayıcısının süresinin dolduğunu belirlemek için geri çağırma işlevi parametresi kullanılır.

Örnek 6.1 iki ayrı geri çağırma işlevi kullandı; biri tek seferlik zamanlayıcı tarafından, diğeri otomatik yeniden yükleme zamanlayıcısı tarafından kullanılıyordu. Örnek 6.2, Örnek 6.1 tarafından oluşturulana benzer işlevsellik oluşturur, ancak her iki yazılım zamanlayıcısına da tek bir geri çağırma işlevi atar.

Örnek 6.2 tarafından kullanılan `main()` işlevi, Örnek 6.1'de kullanılan `main()` işleviyle neredeyse aynıdır. Tek fark, yazılım zamanlayıcılarının oluşturulduğu yerdir. Bu fark, her iki zamanlayıcı için de geri çağırma işlevi olarak `prvTimerCallback()`'ın kullanıldığı Liste 6.10'da gösterilmektedir.

```
/* Tek seferlik yazılım zamanlayıcısını oluşturun, tanıtıcıyı
   xOneShotTimer'da saklayın. */
xOneShotTimer = xTimerCreate( "OneShot",
                              mainONE_SHOT_TIMER_PERIOD,
                              pdFALSE,
                              /* Zamanlayıcının ID'si NULL olarak başlatılır.
*/
                              NULL,
                              /* prvTimerCallback() her iki zamanlayıcı
tarafından kullanılır. */
                              prvTimerCallback );

/* Otomatik yeniden yükleme yazılım zamanlayıcısını oluşturun, tanıtıcıyı
   xAutoReloadTimer'da saklayın. */
xAutoReloadTimer = xTimerCreate( "AutoReload",
                                 mainAUTO_RELOAD_TIMER_PERIOD,
                                 pdTRUE,
                                 /* Zamanlayıcının ID'si NULL olarak
başlatılır. */
                                 NULL,
                                 /* prvTimerCallback() her iki zamanlayıcı
tarafından kullanılır. */
                                 prvTimerCallback );
```

Liste 6.10 *Örnek 6.2'de kullanılan zamanlayıcıları oluşturma*

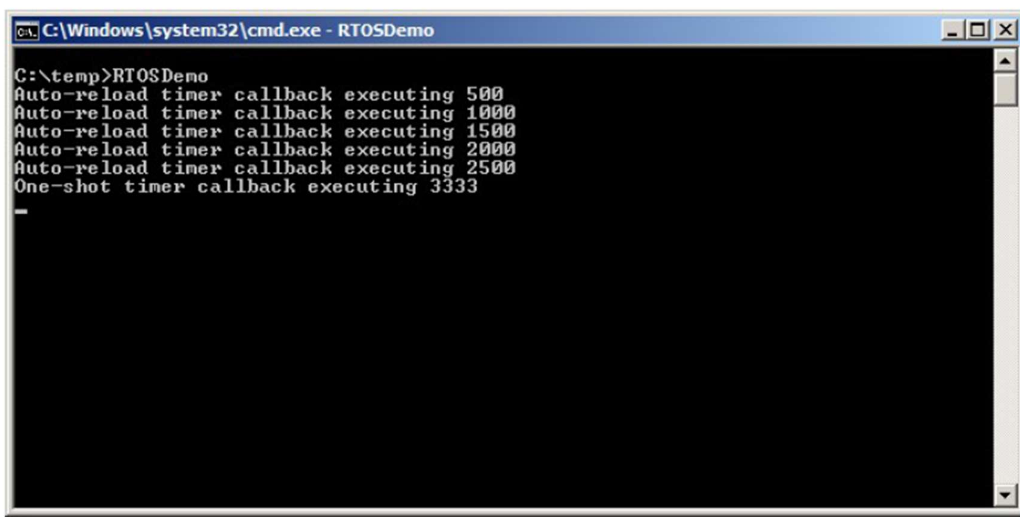
`prvTimerCallback()` her iki zamanlayıcının süresi dolduğunda yürütülecektir. `prvTimerCallback()`'ın uygulaması, tek seferlik zamanlayıcının mı yoksa otomatik yeniden yükleme zamanlayıcısının mı süresinin dolduğunu belirlemek için işlevin parametresini kullanır. `prvTimerCallback()` ayrıca yazılım zamanlayıcı ID'sinin zamanlayıcıya özgü depolama olarak nasıl kullanılacağını da gösterir; her yazılım zamanlayıcısı kaç kez süresinin dolduğunu kendi ID'sinde tutar ve otomatik yeniden yükleme zamanlayıcısı beşinci kez yürütüldüğünde kendini durdurmak için bu sayacı kullanır. `prvTimerCallback()`'ın uygulaması Liste 6.11'de gösterilmektedir.

```
static void prvTimerCallback( TimerHandle_t xTimer )
{
```

```
    TickType_t xTimeNow;
    uint32_t ulExecutionCount;

    /* Bu yazılım zamanlayıcısının kaç kez süresinin dolduğunun sayısı
       zamanlayıcının ID'sinde saklanır. ID'yi alın, artırın ve ardından
       yeni ID değeri olarak kaydedin. ID bir void işaretçisidir,
       bu nedenle uint32_t'ye dönüştürülür. */
    ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );
    ulExecutionCount++;
    vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

    /* Geçerli tick sayısını alın. */
    xTimeNow = xTaskGetTickCount();

    /* Tek seferlik zamanlayıcının tanıtıcısı (handle) zamanlayıcı
       oluşturulduğunda xOneShotTimer'da saklanmıştır. Bu işleve
       iletilen tanıtıcıyı xOneShotTimer ile karşılaştırarak tek
       seferlik mi yoksa otomatik yeniden yükleme zamanlayıcısının mı
       süresinin dolduğunu belirleyin, ardından geri çağırmanın
       yürütüldüğü zamanı gösteren bir dize çıktılayın. */
    if( xTimer == xOneShotTimer )
    {
        vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );
    }
    else
    {
        /* xTimer, xOneShotTimer'a eşit değildi, bu nedenle süresi dolan
           otomatik yeniden yükleme zamanlayıcısı olmuş olmalıdır. */
        vPrintStringAndNumber( "Auto-reload timer callback executing",
xTimeNow);

        if( ulExecutionCount == 5 )
        {
            /* Otomatik yeniden yükleme zamanlayıcısını 5 kez yürütüldükten
               sonra durdurun. Bu geri çağırma işlevi RTOS arka plan
               görevinin bağlamında yürütülür, bu nedenle arka plan
               görevini Engellenmiş durumuna yerleştirebilecek işlevler
               çağırmamalıdır. Bu yüzden 0 engelleme süresi kullanılır. */
            xTimerStop( xTimer, 0 );
        }
    }
}
```

Liste 6.11 *Örnek 6.2'de kullanılan zamanlayıcı geri çağırma işlevi*

Örnek 6.2 tarafından üretilen çıktı Şekil 6.8'de gösterilmektedir. Otomatik yeniden yükleme zamanlayıcısının yalnızca beş kez yürütüldüğü görülebilir.



Şekil 6.8 *Örnek 6.2 yürütüldüğünde üretilen çıktı*

6.7 Bir Zamanlayıcının Periyodunu Değiştirme

Her resmi FreeRTOS portu bir veya daha fazla örnek projeyle birlikte sağlanır. Çoğu örnek proje kendi kendini denetler ve projenin durumu hakkında görsel geri bildirim vermek için bir LED kullanılır; kendi kendine denetimler her zaman başarılı olmuşsa LED yavaş yanıp söner, bir kendi kendine denetim başarısız olmuşsa LED hızlı yanıp söner.

Bazı örnek projeler kendi kendine denetimleri bir görevde gerçekleştirir ve LED'in yanıp sönme hızını kontrol etmek için vTaskDelay() işlevini kullanır. Diğer örnek projeler kendi kendine denetimleri bir yazılım zamanlayıcı geri çağırma işlevinde gerçekleştirir ve LED'in yanıp sönme hızını kontrol etmek için zamanlayıcının periyodunu kullanır.

6.7.1 xTimerChangePeriod() API İşlevi

Bir yazılım zamanlayıcısının periyodu `xTimerChangePeriod()` işlevi kullanılarak değiştirilir.

Eğer `xTimerChangePeriod()` halihazırda çalışan bir zamanlayıcının periyodunu değiştirmek için kullanılırsa, zamanlayıcı yeni periyot değerini kullanarak sona erme zamanını yeniden hesaplar. Yeniden hesaplanan sona erme zamanı `xTimerChangePeriod()`'ın çağrıldığı zamana görelidir, zamanlayıcının başlangıçta başlatıldığı zamana göre değil.

Eğer `xTimerChangePeriod()` Uyku (Dormant) durumundaki (çalışmayan) bir zamanlayıcının periyodunu değiştirmek için kullanılırsa, zamanlayıcı bir sona erme zamanı hesaplar ve Çalışıyor durumuna geçer (zamanlayıcı çalışmaya başlar).

***Not:** Bir kesme hizmet rutininden asla xTimerChangePeriod() çağırmayın. Bunun yerine kesme güvenli sürümü olan xTimerChangePeriodFromISR() kullanılmalıdır.*

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xTicksToWait );
```

***Liste 6.12** xTimerChangePeriod() API işlev prototipi*

xTimerChangePeriod() parametreleri ve dönüş değeri

- `xTimer`
  Yeni bir periyot değeriyle güncellenen yazılım zamanlayıcısının tanıtıcısı (handle).
- `xTimerPeriodInTicks`
  Yazılım zamanlayıcısı için tick cinsinden belirtilen yeni periyot. `pdMS_TO_TICKS()` makrosu milisaniye cinsinden belirtilen bir süreyi tick cinsinden belirtilen bir süreye dönüştürmek için kullanılabilir.

- `xTicksToWait`
  `xTimerChangePeriod()` arka plan görevine 'periyodu değiştir' komutunu göndermek için zamanlayıcı komut kuyruğunu kullanır. `xTicksToWait` kuyruk zaten dolu ise çağıran görevin zamanlayıcı komut kuyruğunda yer açılması için Engellenmiş durumda kalması gereken maksimum süreyi belirtir.

- Döndürülen değer
  İki olası dönüş değeri vardır:
  - `pdPASS` — Veri zamanlayıcı komut kuyruğuna başarıyla gönderildiğinde döndürülür.
  - `pdFAIL` — 'Periyodu değiştir' komutu kuyruk zaten dolu olduğu için zamanlayıcı komut kuyruğuna yazılamadığında döndürülür.

Liste 6.13, FreeRTOS örneklerinin bir yazılım zamanlayıcı geri çağırma işlevinde kendi kendine denetim işlevselliği içermesinin, bir kendi kendine denetim başarısız olursa LED'in yanıp sönme hızını artırmak için xTimerChangePeriod() kullanımını nasıl gösterdiğini göstermektedir. Kendi kendine denetimleri gerçekleştiren yazılım zamanlayıcısına 'denetim zamanlayıcısı' denir.

```
/* Denetim zamanlayıcısı 3000 milisaniye periyotla oluşturulur ve LED her
   3 saniyede bir yanıp söner. Kendi kendine denetim işlevselliği beklenmedik
   bir durum tespit ederse, denetim zamanlayıcısının periyodu yalnızca
   200 milisaniyeye değiştirilir ve çok daha hızlı bir yanıp sönme hızı oluşur.
*/
const TickType_t xHealthyTimerPeriod = pdMS_TO_TICKS( 3000 );
const TickType_t xErrorTimerPeriod = pdMS_TO_TICKS( 200 );

/* Denetim zamanlayıcısı tarafından kullanılan geri çağırma işlevi. */
static void prvCheckTimerCallbackFunction( TimerHandle_t xTimer )
{
    static BaseType_t xErrorDetected = pdFALSE;

    if( xErrorDetected == pdFALSE )
    {
```

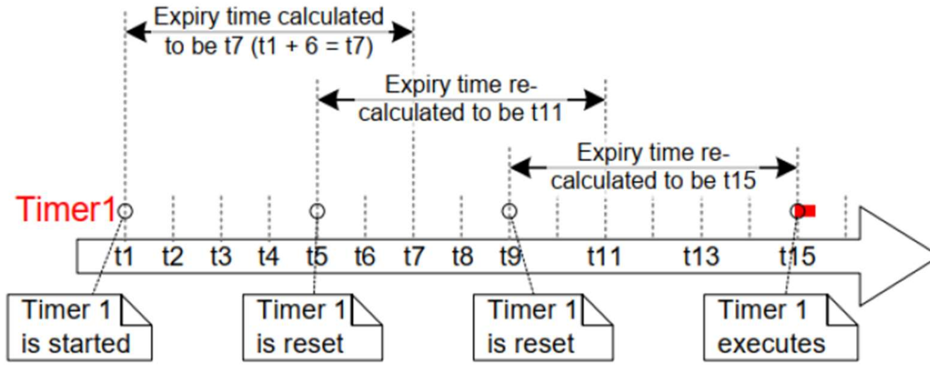
```
        /* Henüz hata tespit edilmedi. Kendi kendine denetim işlevini tekrar
           çalıştırın. İşlev, örnek tarafından oluşturulan her göreve kendi
           durumunu bildirmesini sorar ve ayrıca tüm görevlerin hâlâ
           çalışıyor olduğunu (ve durumlarını doğru bildirebilecek durumda
           olduğunu) kontrol eder. */
        if( CheckTasksAreRunningWithoutError() == pdFAIL )
        {
            /* Bir veya daha fazla görev beklenmedik bir durum bildirdi.
               Bir hata oluşmuş olabilir. Denetim zamanlayıcısının periyodunu
               bu geri çağırma işlevinin yürütülme hızını artırmak ve böylece
               LED'in yanıp sönme hızını da artırmak için azaltın. Bu geri
               çağırma işlevi RTOS arka plan görevinin bağlamında yürütülür,
               bu nedenle arka plan görevinin Engellenmiş durumuna asla
               girmemesini sağlamak için 0 engelleme süresi kullanılır. */
            xTimerChangePeriod(
                xTimer,             /* Güncellenen zamanlayıcı */
                xErrorTimerPeriod,  /* Zamanlayıcı için yeni periyot */
                0 );                /* Bu komutu gönderirken engelleme yapma */
        }

        /* Bir hatanın zaten tespit edildiğini kaydedin. */
        xErrorDetected = pdTRUE;
    }

    /* LED'i değiştirin. LED'in değişme hızı bu işlevin ne sıklıkta
       çağrıldığına bağlıdır ve bu denetim zamanlayıcısının periyodu
       tarafından belirlenir. CheckTasksAreRunningWithoutError() hiç
       pdFAIL döndürmüşse zamanlayıcının periyodu 3000ms'den
       yalnızca 200ms'ye düşürülmüş olacaktır. */
    ToggleLED();
}
```

Liste 6.13 *xTimerChangePeriod() kullanımı*

6.8 Bir Yazılım Zamanlayıcısını Sıfırlama

Bir yazılım zamanlayıcısını sıfırlamak, zamanlayıcıyı yeniden başlatmak anlamına gelir; zamanlayıcının sona erme zamanı, zamanlayıcının başlangıçta başlatıldığı zamana göre değil, zamanlayıcının sıfırlandığı zamana göre yeniden hesaplanır.

Bu, Şekil 6.9 ile gösterilmektedir; Şekil 6.9, periyodu 6 olan bir zamanlayıcının başlatılmasını, ardından geri çağırma işlevini yürütmeden önce iki kez sıfırlanmasını göstermektedir.



Şekil 6.9 *Periyodu 6 tick olan bir yazılım zamanlayıcısının başlatılması ve sıfırlanması*

Şekil 6.9'a atıfla:

• Zamanlayıcı 1, t1 zamanında başlatılır. Periyodu 6'dır, bu nedenle geri çağırma işlevini yürüteceği zaman başlangıçta t7 olarak hesaplanır, yani başlatıldıktan 6 tick sonra.

• Zamanlayıcı 1, t7 zamanına ulaşılmadan önce, yani süresi dolmadan ve geri çağırma işlevini yürütmeden önce sıfırlanır. Zamanlayıcı 1, t5 zamanında sıfırlanır, bu nedenle geri çağırma işlevini yürüteceği zaman t11 olarak yeniden hesaplanır, yani sıfırlandıktan 6 tick sonra.

• Zamanlayıcı 1, t11 zamanından önce tekrar sıfırlanır, yani yine süresi dolmadan ve geri çağırma işlevini yürütmeden önce. Zamanlayıcı 1, t9 zamanında sıfırlanır, bu nedenle geri çağırma işlevini yürüteceği zaman t15 olarak yeniden hesaplanır, yani en son sıfırlandıktan 6 tick sonra.

• Zamanlayıcı 1 tekrar sıfırlanmaz, bu nedenle t15 zamanında süresi dolar ve geri çağırma işlevi buna göre yürütülür.

6.8.1 xTimerReset() API İşlevi

Bir zamanlayıcı `xTimerReset()` API işlevi kullanılarak sıfırlanır. `xTimerReset()` Uyku (Dormant) durumundaki bir zamanlayıcıyı başlatmak için de kullanılabilir.

Not: *Bir kesme hizmet rutininden asla xTimerReset() çağırmayın. Bunun yerine kesme güvenli sürümü olan xTimerResetFromISR() kullanılmalıdır.*

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Liste 6.14 *xTimerReset() API işlev prototipi*

xTimerReset() parametreleri ve dönüş değeri

- `xTimer`
  Sıfırlanan veya başlatılan yazılım zamanlayıcısının tanıtıcısı (handle). Tanıtıcı, yazılım zamanlayıcısını oluşturmak için kullanılan `xTimerCreate()` çağrısından döndürülmüş olacaktır.

- `xTicksToWait`
  `xTimerChangePeriod()` daemon görevine 'sıfırla' komutunu göndermek için zamanlayıcı komut kuyruğunu kullanır. `xTicksToWait`, kuyruk zaten dolu olması durumunda çağıran görevin zamanlayıcı komut kuyruğunda yer açılması için Engellenmiş durumda kalması gereken maksimum süreyi belirtir.
  `xTimerReset()`, `xTicksToWait` sıfırsa ve zamanlayıcı komut kuyruğu zaten doluysa anında geri döner.
  `INCLUDE_vTaskSuspend` `FreeRTOSConfig.h` dosyasında 1 olarak ayarlanmışsa, `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak, çağıran görevin zamanlayıcı komut kuyruğunda yer açılması için süresiz olarak (zaman aşımı olmadan) Engellenmiş durumda kalmasına neden olur.

- **Dönüş değeri**
  İki olası dönüş değeri vardır:
  - `pdPASS`
    `pdPASS` yalnızca veriler zamanlayıcı komut kuyruğuna başarıyla gönderildiğinde döndürülür.
    Bir engelleme süresi belirtilmişse (`xTicksToWait` sıfır değilse), çağıran görevin işlev dönmeden önce zamanlayıcı komut kuyruğunda yer açılmasını beklemek için Engellenmiş durumuna yerleştirilmiş olması mümkündür, ancak engelleme süresi dolmadan veriler zamanlayıcı komut kuyruğuna başarıyla yazılmıştır.
  - `pdFAIL`
    `pdFAIL`, kuyruk zaten dolu olduğu için 'sıfırla' komutu zamanlayıcı komut kuyruğuna yazılamadığında döndürülür.
    Bir engelleme süresi belirtilmişse (`xTicksToWait` sıfır değilse), çağıran görev daemon görevinin kuyrukta yer açmasını beklemek için Engellenmiş durumuna yerleştirilmiş olacaktır, ancak belirtilen engelleme süresi bu gerçekleşmeden önce dolmuştur.

Örnek 6.3 Bir yazılım zamanlayıcısını sıfırlama

Bu örnek, bir cep telefonundaki arka ışığın davranışını simüle eder. Arka ışık:
- Bir tuşa basıldığında açılır.
- Belirli bir süre içinde daha fazla tuşa basıldığı sürece açık kalır.
- Belirli bir süre içinde hiçbir tuşa basılmazsa otomatik olarak kapanır.

Bu davranışı uygulamak için tek seferlik (one-shot) bir yazılım zamanlayıcısı kullanılır:
- [Simüle edilen] arka ışık, bir tuşa basıldığında açılır ve yazılım zamanlayıcısının geri çağırma işlevinde kapatılır.
- Yazılım zamanlayıcısı, her tuşa basıldığında sıfırlanır.
- Arka ışığın kapanmasını önlemek için bir tuşa basılması gereken süre, bu nedenle yazılım zamanlayıcısının periyoduna eşittir; yazılım zamanlayıcısı

zamanlayıcı süresi dolmadan bir tuşa basılarak sıfırlanmazsa, zamanlayıcının geri çağırma işlevi yürütülür ve arka ışık kapatılır.

xSimulatedBacklightOn değişkeni arka ışık durumunu tutar. xSimulatedBacklightOn arka ışık açık olduğunu belirtmek için pdTRUE olarak, kapalı olduğunu belirtmek için pdFALSE olarak ayarlanır.

Yazılım zamanlayıcı geri çağırma işlevi Liste 6.15'te gösterilmektedir.

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow = xTaskGetTickCount();

    /* Arka ışık zamanlayıcısının süresi doldu, arka ışığı kapatın. */
    xSimulatedBacklightOn = pdFALSE;

    /* Arka ışığın kapatıldığı zamanı yazdırın. */
    vPrintStringAndNumber(
            "Timer expired, turning backlight OFF at time\t\t", xTimeNow );
}
```

Liste 6.15 *Örnek 6.3'te kullanılan tek seferlik zamanlayıcının geri çağırma işlevi*

Örnek 6.3, klavyeyi yoklamak (poll) için bir görev oluşturur. Görev, Liste 6.16'da gösterilmektedir, ancak bir sonraki paragrafta açıklanan nedenlerle, Liste 6.16 optimal bir tasarımı temsil etmeyi amaçlamamaktadır.

[^11]: Windows konsoluna yazdırma ve Windows konsolundan tuş okuma, her ikisi de Windows sistem çağrılarının yürütülmesine neden olur. Windows konsolu, diskler veya TCP/IP yığını kullanımı dahil olmak üzere Windows sistem çağrıları, FreeRTOS Windows portunun davranışını olumsuz etkileyebilir ve normalde bundan kaçınılmalıdır.*

FreeRTOS kullanmak, uygulamanızın olay güdümlü olmasını sağlar. Olay güdümlü tasarımlar işlem süresini çok verimli kullanır, çünkü işlem süresi yalnızca bir olay gerçekleştiğinde kullanılır ve gerçekleşmemiş olaylar için yoklama yapılarak işlem süresi boşa harcanmaz. Örnek 6.3, FreeRTOS Windows portunu kullanırken klavye kesmelerini işlemenin pratik olmaması nedeniyle olay güdümlü yapılamamıştır, bu nedenle çok daha az verimli olan yoklama tekniği kullanılmak zorunda kalınmıştır. Liste 6.16 bir kesme hizmet rutini olsaydı, xTimerResetFromISR() xTimerReset() yerine kullanılırdı.

```
static void vKeyHitTask( void *pvParameters )
{
    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );
    TickType_t xTimeNow;

    vPrintString( "Press a key to turn the backlight on.\r\n" );

    /* İdeal olarak bir uygulama olay güdümlü olur ve tuşa basmaları
       işlemek için bir kesme kullanırdı. FreeRTOS Windows portunu
```

```
        kullanırken klavye kesmelerini kullanmak pratik değildir, bu
        nedenle bu görev tuşa basma yoklaması yapmak için kullanılır. */
    for( ;; )
    {
        /* Bir tuşa basıldı mı? */
        if( _kbhit() != 0 )
        {
            /* Bir tuşa basıldı. Zamanı kaydedin. */
            xTimeNow = xTaskGetTickCount();

            if( xSimulatedBacklightOn == pdFALSE )
            {
                /* Arka ışık kapalıydı, açın ve açıldığı zamanı
                   yazdırın. */
                xSimulatedBacklightOn = pdTRUE;
                vPrintStringAndNumber(
                        "Key pressed, turning backlight ON at time\t",
                        xTimeNow );
            }
            else
            {
                /* Arka ışık zaten açıktı, zamanlayıcının sıfırlanmak
                   üzere olduğunu ve sıfırlandığı zamanı yazdırın. */
                vPrintStringAndNumber(
                        "Key pressed, resetting software timer at time\t",
                        xTimeNow );
            }

            /* Yazılım zamanlayıcısını sıfırlayın. Arka ışık daha önce
               kapalıydıysa, bu çağrı zamanlayıcıyı başlatır. Arka ışık
               daha önce açıksa, bu çağrı zamanlayıcıyı yeniden
               başlatır. Gerçek bir uygulama bir kesmede tuşa basmaları
               okuyabilir. Bu işlev bir kesme hizmet rutini olsaydı
               xTimerReset() yerine xTimerResetFromISR() kullanılmalıdır. */
            xTimerReset( xBacklightTimer, xShortDelay );

            /* Basılan tuşu okuyun ve atın – bu basit örnek için
               gerekli değildir. */
            ( void ) _getch();
        }
    }
}
```

Liste 6.16 *Örnek 6.3'te yazılım zamanlayıcısını sıfırlamak için kullanılan görev*

Örnek 6.3 yürütüldüğünde üretilen çıktı Şekil 6.10'da gösterilmektedir. Şekil 6.10'a atıfla:

- İlk tuşa basma, tick sayısı 812 olduğunda gerçekleşmiştir. O anda arka ışık açılmış ve tek seferlik zamanlayıcı başlatılmıştır.
- Daha sonraki tuşa basmalar, tick sayısı 1813, 3114, 4015 ve 5016 olduğunda gerçekleşmiştir. Tüm bu tuşa basmalar, zamanlayıcının süresi dolmadan zamanlayıcının sıfırlanmasına neden olmuştur.
- Zamanlayıcının süresi, tick sayısı 10016 olduğunda dolmuştur. O anda arka ışık kapatılmıştır.

Şekil 6.10 *Örnek 6.3 yürütüldüğünde üretilen çıktı*

Şekil 6.10'da zamanlayıcının 5000 tick periyoduna sahip olduğu görülebilir; arka ışık, bir tuşa en son basıldıktan tam olarak 5000 tick sonra, yani zamanlayıcı en son sıfırlandıktan 5000 tick sonra kapatılmıştır.

Örnek 6.3 *Bir yazılım zamanlayıcısını sıfırlama*

Bu örnek, bir cep telefonundaki arka ışığın davranışını simüle eder. Arka ışık:
• Bir tuşa basıldığında açılır.
• Belirli bir süre içinde başka tuşlara basıldığı sürece açık kalır.
• Belirli bir süre içinde hiçbir tuşa basılmazsa otomatik olarak kapanır.
Bu davranışı uygulamak için tek seferlik bir yazılım zamanlayıcısı kullanılır:
• [Simüle edilen] arka ışık bir tuşa basıldığında açılır ve yazılım zamanlayıcısının geri çağırma işlevinde kapatılır.
• Yazılım zamanlayıcısı her tuşa basıldığında sıfırlanır.
• Arka ışığın kapatılmasını önlemek için bir tuşa basılması gereken süre, bu nedenle yazılım zamanlayıcısının periyoduna eşittir; zamanlayıcının süresi dolmadan önce bir tuş basımı ile sıfırlanmazsa, zamanlayıcının geri çağırma işlevi yürütülür ve arka ışık kapatılır.
xSimulatedBacklightOn değişkeni arka ışık durumunu tutar. xSimulatedBacklightOn arka ışığın açık olduğunu belirtmek için pdTRUE, kapalı olduğunu belirtmek için pdFALSE olarak ayarlanır.

Yazılım zamanlayıcı geri çağırma işlevi Liste 6.15'te gösterilmektedir.

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow = xTaskGetTickCount();

    /* Arka ışık zamanlayıcısının süresi doldu, arka ışığı kapatın. */
    xSimulatedBacklightOn = pdFALSE;

    /* Arka ışığın kapatıldığı zamanı yazdırın. */
    vPrintStringAndNumber(
            "Timer expired, turning backlight OFF at time\t\t", xTimeNow );
}
```

Liste 6.15 *Örnek 6.3'te kullanılan tek seferlik zamanlayıcının geri çağırma işlevi*

Örnek 6.3, klavyeyi yoklamak (poll) için bir görev oluşturur. Görev, Liste 6.16'da gösterilmektedir.

```
static void vKeyHitTask( void *pvParameters )
{
    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );
    TickType_t xTimeNow;

    vPrintString( "Press a key to turn the backlight on.\r\n" );

    /* İdeal olarak bir uygulama olay güdümlü olur ve tuş basımlarını
       işlemek için bir kesme kullanırdı. FreeRTOS Windows portunu
       kullanırken klavye kesmelerini kullanmak pratik değildir,
```

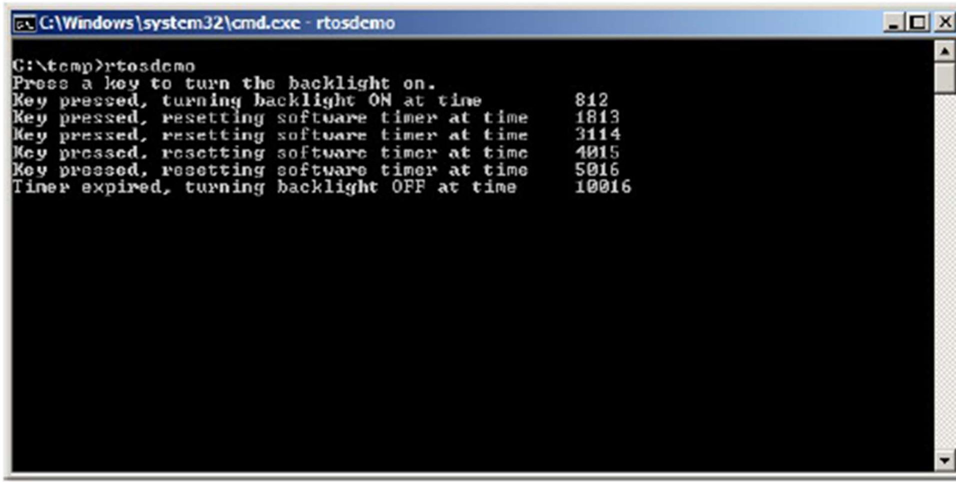
```
        bu nedenle bu görev bir tuş basımını yoklamak için kullanılır. */
    for( ;; )
    {
        /* Bir tuşa basıldı mı? */
        if( _kbhit() != 0 )
        {
            /* Bir tuşa basıldı. Zamanı kaydedin. */
            xTimeNow = xTaskGetTickCount();

            if( xSimulatedBacklightOn == pdFALSE )
            {
                /* Arka ışık kapalıydı, açın ve açıldığı
                   zamanı yazdırın. */
                xSimulatedBacklightOn = pdTRUE;
                vPrintStringAndNumber(
                    "Key pressed, turning backlight ON at time\t\t",
                    xTimeNow );
            }
            else
            {
                /* Arka ışık zaten açıktı, sıfırlanmak üzere olduğunu
                   ve sıfırlandığı zamanı belirten bir mesaj yazdırın. */
                vPrintStringAndNumber(
                    "Key pressed, resetting software timer at time\t\t",
                    xTimeNow );
            }

            /* Yazılım zamanlayıcısını sıfırlayın. Arka ışık daha önce
               kapalıysa bu çağrı zamanlayıcıyı başlatır. Arka ışık
               daha önce açıksa bu çağrı zamanlayıcıyı yeniden başlatır.
               Gerçek bir uygulama bir kesmede tuş basımlarını okuyabilir.
               Bu işlev bir kesme hizmet rutini olsaydı
               xTimerReset() yerine xTimerResetFromISR() kullanılmalıdır. */
            xTimerReset( xBacklightTimer, xShortDelay );

            /* Basılan tuşu okuyup atın – bu basit örnek için
               gerekli değildir. */
            ( void ) _getch();
        }
    }
}
```

Liste 6.16 *Örnek 6.3'te yazılım zamanlayıcısını sıfırlamak için kullanılan görev*

Örnek 6.3 yürütüldüğünde üretilen çıktı Şekil 6.10'da gösterilmektedir. Şekil 6.10'a atıfla:

- İlk tuş basımı tick sayısı 812 olduğunda gerçekleşti. O zaman arka ışık açıldı ve tek seferlik zamanlayıcı başlatıldı.
- Tick sayısı 1813, 3114, 4015 ve 5016 olduğunda başka tuş basımları gerçekleşti. Tüm bu tuş basımları zamanlayıcının süresinin dolmasından önce zamanlayıcının sıfırlanmasıyla sonuçlandı.

• Zamanlayıcı tick sayısı 10016 olduğunda süresinin doldu. O zaman arka ışık kapatıldı.



Şekil 6.10 *Örnek 6.3 yürütüldüğünde üretilen çıktı*

Şekil 6.10'da zamanlayıcının periyodunun 5000 tick olduğu görülebilir; arka ışık bir tuşa en son basıldıktan tam 5000 tick sonra, yani zamanlayıcının en son sıfırlanmasından 5000 tick sonra kapatılmıştır.

7 Kesme Yönetimi (Interrupt Management)

7.1 Giriş

7.1.1 Olaylar (Events)

Gömülü gerçek zamanlı sistemler, ortamdan kaynaklanan olaylara tepki olarak eylemler gerçekleştirmelidir. Örneğin, bir Ethernet çevre biriminden gelen bir paket (olay) bir TCP/IP yığınına iletilmek (eylem) üzere işlenmelidir. Basit olmayan (non-trivial) sistemler, birden fazla kaynaktan gelen olaylara hizmet vermek zorundadır ve tüm bu olayların farklı işlem yükleri ve yanıt süresi gereksinimleri olacaktır. Her durumda, en iyi olay işleme uygulama stratejisine ilişkin bir karar verilmelidir:

• Olay nasıl tespit edilmelidir? Kesmeler normalde kullanılır, ancak girişler yoklanabilir (polling) de.

• Kesmeler kullanıldığında, ne kadar işlem kesme hizmet rutini (ISR) içinde yapılmalı, ne kadarı dışında? Normalde her ISR'yi mümkün olduğunca kısa tutmak arzu edilir.

• Olaylar ana (ISR olmayan) koda nasıl iletilir ve bu kod potansiyel olarak eş zamanlı olmayan olayların işlenmesini en iyi şekilde barındıracak şekilde nasıl yapılandırılabilir?

FreeRTOS, uygulama tasarımcısına belirli bir olay işleme stratejisi dayatmaz, ancak seçilen stratejinin basit ve sürdürülebilir bir şekilde uygulanmasını sağlayan özellikler sunar.

Bir görevin önceliği ile bir kesmenin önceliği arasında ayrım yapmak önemlidir:

• Bir görev, FreeRTOS'un üzerinde çalıştığı donanımla ilişkisi olmayan bir yazılım özelliğidir. Bir görevin önceliği uygulama yazarı tarafından yazılımda atanır ve bir yazılım algoritması (çizgeleyici) hangi görevin Çalışıyor durumuna yerleştirileceğine karar verir.

• Yazılımda yazılmış olmasına rağmen, bir kesme hizmet rutini bir donanım özelliğidir çünkü donanım hangi kesme hizmet rutininin çalışacağını ve ne zaman çalışacağını kontrol eder. Görevler yalnızca hiçbir ISR çalışmadığında yürütülür, bu nedenle en düşük öncelikli kesme bile en yüksek öncelikli görevi kesintiye uğratacaktır ve bir görevin bir ISR'yi önceleyen (pre-empt eden) bir yolu yoktur.

FreeRTOS'un çalışacağı tüm mimariler kesme işleme yeteneğine sahiptir, ancak kesme girişi, kesme önceliği ataması ve kesme işlemeye ilişkin ayrıntılar mimariler arasında farklılık gösterir.

7.1.2 Kapsam

Bu bölüm şunları kapsar:

• Bir kesme hizmet rutini içinden hangi FreeRTOS API işlevlerinin kullanılabileceği.
• Kesme işlemeyi bir göreve erteleme yöntemleri.
• İkili (binary) ve sayma (counting) semaforlarının nasıl oluşturulacağı ve kullanılacağı.
• İkili ve sayma semaforları arasındaki farklar.
• Bir kesme hizmet rutinine ve bir kesme hizmet rutininden veri aktarmak için bir kuyruğun nasıl kullanılacağı.
• Bazı FreeRTOS portlarında bulunan kesme iç içe geçme (nesting) modeli.

7.2 Bir ISR'den FreeRTOS API'sini Kullanma

7.2.1 Kesme Güvenli (Interrupt Safe) API

Bir kesme hizmet rutininden (ISR) FreeRTOS API işlevinin sağladığı işlevselliği kullanmak genellikle gereklidir, ancak birçok FreeRTOS API işlevi bir ISR içinde geçerli olmayan eylemler gerçekleştirir. Bunların en dikkat çekeni, API işlevini çağıran görevi Engellenmiş durumuna yerleştirmektir — bir API işlevi bir ISR'den çağrılıyorsa, bir görevden çağrılmıyordur, bu nedenle Engellenmiş durumuna yerleştirilebilecek çağıran bir görev yoktur. FreeRTOS bu sorunu bazı API işlevlerinin iki versiyonunu sağlayarak çözer; biri görevlerden kullanım için, diğeri ISR'lerden kullanım için. ISR'lerden kullanılmak üzere tasarlanan işlevlerin adlarına "FromISR" eklenir.

Not: Adında "FromISR" bulunmayan bir FreeRTOS API işlevini asla bir ISR'den çağırmayın.

7.2.2 Ayrı Bir Kesme Güvenli API Kullanmanın Faydaları

Kesmelerde kullanılmak üzere ayrı bir API'ye sahip olmak, görev kodunun daha verimli, ISR kodunun daha verimli ve kesme girişinin daha basit olmasını sağlar. Nedenini görmek için, alternatif çözümü düşünün; bu çözüm, hem bir görevden hem de bir ISR'den çağrılabilecek her API işlevinin tek bir versiyonunu sağlamak olurdu. Bir API işlevinin aynı versiyonu hem bir görevden hem de bir ISR'den çağrılabilseydi:

• API işlevlerinin bir görevden mi yoksa bir ISR'den mi çağrıldığını belirlemek için ek mantık gerekir. Ek mantık, işlev boyunca yeni yollar ekleyerek işlevleri daha uzun, daha karmaşık ve test edilmesi daha zor hale getirir.

• Bazı API işlev parametreleri işlev bir görevden çağrıldığında gereksiz olurken, diğerleri işlev bir ISR'den çağrıldığında gereksiz olur.

• Her FreeRTOS portu, yürütme bağlamını (görev veya ISR) belirlemek için bir mekanizma sağlamalıdır.

• Yürütme bağlamını (görev veya ISR) belirlemenin kolay olmadığı mimarilerde, yürütme bağlamının yazılım tarafından sağlanmasına izin veren ek, savurgan, daha karmaşık ve standart olmayan kesme giriş kodu gerekir.

7.2.3 Ayrı, Kesme Güvenli Bir API Kullanmanın Dezavantajları

Bazı API işlevlerinin iki sürümüne sahip olmak hem görevlerin hem de ISR'lerin daha verimli olmasını sağlar, ancak yeni bir soruna yol açar; bazen FreeRTOS API'sinin bir parçası olmayan, ancak FreeRTOS API'sini kullanan bir işlevi hem bir görevden hem de bir ISR'den çağırmak gerekir.

Bu, normalde yalnızca üçüncü taraf (third party) kodlar entegre edilirken sorun olur, çünkü yazılımın tasarımının uygulama yazarının kontrolü dışında olduğu tek zaman budur. Eğer bu bir sorun haline gelirse, sorun aşağıdaki tekniklerden biri kullanılarak aşılabilir:

- Kesme işlemini bir göreve erteleyin (defer), böylece API işlevi yalnızca bir görev bağlamından çağrılmış olur.
- Kesme yuvalamasını (interrupt nesting) destekleyen bir FreeRTOS portu kullanıyorsanız, API işlevinin "FromISR" ile biten sürümünü kullanın, çünkü o sürüm hem görevlerden hem de ISR'lerden çağrılabilir. (Tersi doğru değildir, "FromISR" ile bitmeyen API işlevleri bir ISR'den çağrılmamalıdır.)
- Üçüncü taraf kodu genellikle, işlevin hangi bağlamdan (görev veya kesme) çağrıldığını test etmek için uygulanabilen ve ardından bağlam için uygun olan API işlevini çağıran bir RTOS soyutlama katmanı (RTOS abstraction layer) içerir.

7.2.4 xHigherPriorityTaskWoken Parametresi

Bu bölüm xHigherPriorityTaskWoken parametresinin kavramını tanıtır. Bu bölümü henüz tam olarak anlamamazsanız endişelenmeyin, çünkü pratik örnekler ilerideki bölümlerde sunulmaktadır.

Bir kesme tarafından bir bağlam değişikliği (context switch) yapılırsa, kesme çıktığında çalışan görev, kesmeye girildiğinde çalışan görevden farklı olabilir — kesme bir görevi kesintiye uğratmış, ancak farklı bir göreve dönmüş olacaktır.
Bazı FreeRTOS API işlevleri bir görevi Engellenmiş durumdan Hazır durumuna taşıyabilir. Bu, xQueueSendToBack() gibi işlevlerde zaten görülmüştür; bu işlev, ilgili kuyrukta verinin kullanılabilir olmasını bekleyen Engellenmiş durumdaki bir görev varsa görevi engellemeden çıkarır.

Bir FreeRTOS API işlevi tarafından engeli kaldırılan bir görevin önceliği, Çalışıyor durumundaki görevin önceliğinden yüksekse, FreeRTOS çizgeleme politikasına göre daha yüksek öncelikli göreve geçiş yapılmalıdır. Daha yüksek öncelikli göreve geçişin gerçekleştiği zaman, API işlevinin çağrıldığı bağlama bağlıdır:

• API işlevi bir görevden çağrıldıysa:
FreeRTOSConfig.h'da configUSE_PREEMPTION 1 olarak ayarlanmışsa, daha yüksek öncelikli göreve geçiş API işlevi içinde otomatik olarak gerçekleşir, yani API işlevi çıkmadan önce.

• API işlevi bir kesmeden çağrıldıysa:

Bir kesme içinde daha yüksek öncelikli bir göreve otomatik geçiş gerçekleşmez. Bunun yerine, bir bağlam değişikliğinin yapılması gerektiğini uygulama yazarına bildirmek için bir değişken ayarlanır. Kesme güvenli API işlevleri ("FromISR" ile bitenler) bu amaç için kullanılan pxHigherPriorityTaskWoken adlı bir işaretçi parametresine sahiptir.

Bir bağlam değişikliği yapılması gerekiyorsa, kesme güvenli API işlevi *pxHigherPriorityTaskWoken'ı pdTRUE olarak ayarlayacaktır. Bunun gerçekleştiğini tespit edebilmek için, pxHigherPriorityTaskWoken tarafından işaret edilen değişken ilk kullanılmadan önce pdFALSE olarak başlatılmalıdır.

Uygulama yazarı ISR'den bir bağlam değişikliği talep etmemeyi tercih ederse, daha yüksek öncelikli görev, çizgeleyicinin bir sonraki çalışmasına kadar (en kötü durumda bir sonraki tick kesmesinde olacaktır) Hazır durumunda kalacaktır.

FreeRTOS API işlevleri yalnızca *pxHighPriorityTaskWoken'ı pdTRUE olarak ayarlayabilir. Bir ISR birden fazla FreeRTOS API işlevi çağırıyorsa, aynı değişken her API işlev çağrısında pxHigherPriorityTaskWoken parametresi olarak geçirilebilir ve değişkenin yalnızca ilk kullanılmadan önce pdFALSE olarak başlatılması gerekir. Bağlam değişikliklerinin API işlevinin kesme güvenli versiyonu içinde otomatik olarak gerçekleşmemesinin birkaç nedeni vardır:

• Gereksiz bağlam değişikliklerinden kaçınma
Bir görevin herhangi bir işlem yapması gerekmeden önce bir kesme birden fazla yürütülebilir. Örneğin, kesme güdümlü bir UART tarafından alınan bir dizeyi işleyen bir senaryo düşünün; her karakter alındığında UART ISR'nin göreve geçmesi israf olur çünkü görevin ancak tam dize alındıktan sonra yapacağı işlem olacaktır.

• Yürütme dizisi üzerinde kontrol
Kesmeler düzensiz ve öngörülemeyen zamanlarda oluşabilir. Uzman FreeRTOS kullanıcıları, uygulamalarında belirli noktalarda farklı bir göreve öngörülemeyen bir geçişi geçici olarak önlemek isteyebilir, ancak bu FreeRTOS çizgeleyici kilitleme mekanizması kullanılarak da sağlanabilir.

• Taşınabilirlik
Tüm FreeRTOS portlarında kullanılabilecek en basit mekanizmadır.

• Verimlilik
Daha küçük işlemci mimarilerini hedefleyen portlar yalnızca bir ISR'nin en sonunda bağlam değişikliği talep edilmesine izin verir ve bu kısıtlamayı kaldırmak ek ve daha karmaşık kod gerektirir. Ayrıca, aynı ISR içinde birden fazla bağlam değişikliği talebi oluşturmadan aynı ISR içinde birden fazla FreeRTOS API işlevi çağrısına olanak tanır.

• RTOS tick kesmesinde yürütme
Bu kitapta daha sonra görüleceği gibi, RTOS tick kesmesine uygulama kodu eklemek mümkündür. Tick kesmesi içinde bir bağlam değişikliği denemenin sonucu, kullanılan FreeRTOS portuna bağlıdır. En iyi ihtimalle, çizgeleyiciye gereksiz bir çağrı ile sonuçlanır.

pxHigherPriorityTaskWoken parametresinin kullanımı isteğe bağlıdır. Gerekli değilse, pxHigherPriorityTaskWoken'ı NULL olarak ayarlayın.

7.2.5 portYIELD_FROM_ISR() ve portEND_SWITCHING_ISR() Makroları

`taskYIELD()` bir bağlam geçişi (context switch) istemek için bir görevde çağrılabilen bir makrodur. `portYIELD_FROM_ISR()` ve `portEND_SWITCHING_ISR()` makrolarının her ikisi de `taskYIELD()` makrosunun kesme güvenli sürümleridir. `portYIELD_FROM_ISR()` ve `portEND_SWITCHING_ISR()` makrolarının her ikisi de aynı şekilde kullanılır ve aynı işi yaparlar. Bazı FreeRTOS portları (uyarlamaları) iki makrodan yalnızca birini sağlar. Yeni FreeRTOS portları her iki makroyu da sağlar. Bu kitaptaki örnekler `portYIELD_FROM_ISR()` kullanmaktadır.

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

Liste 7.1 `portEND_SWITCHING_ISR()` makrosu

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

Liste 7.2 `portYIELD_FROM_ISR()` makrosu

Kesme güvenli (interrupt safe) bir API işlevinden aktarılan `xHigherPriorityTaskWoken` parametresi doğrudan `portYIELD_FROM_ISR()` çağrısında parametre olarak kullanılabilir.

Eğer `portYIELD_FROM_ISR()` işlevinin `xHigherPriorityTaskWoken` parametresi `pdFALSE` (sıfır) ise, bağlam geçişi istenmez ve makronun hiçbir etkisi olmaz. Eğer `portYIELD_FROM_ISR()` işlevinin `xHigherPriorityTaskWoken` parametresi `pdFALSE` değilse, bir bağlam geçişi istenir ve Çalışıyor durumundaki görev (Running state task) değişebilir.

Kesme (interrupt) yürütülürken Çalışıyor durumundaki görev değişmiş olsa bile, kesme her zaman Çalışıyor durumundaki göreve geri dönecektir.

Çoğu FreeRTOS portu, `portYIELD_FROM_ISR()` işlevinin bir ISR içinde herhangi bir yerde çağrılmasına izin verir. Birkaç FreeRTOS portu (ağırlıklı olarak daha küçük mimariler için olanlar), `portYIELD_FROM_ISR()` işlevinin yalnızca bir ISR'nin en sonunda (very end) çağrılmasına izin verir.

7.3 Ertelenmiş Kesme İşleme (Deferred Interrupt Processing)

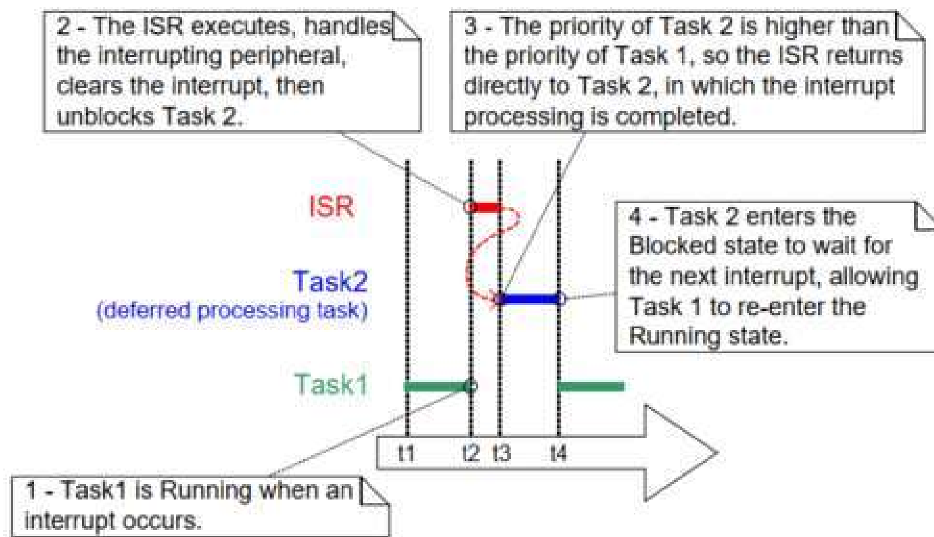
Normalde kesme servis rutinlerinin (ISR'ler) mümkün olduğunca kısa tutulması en iyi uygulama olarak kabul edilir. Bunun nedenleri şunlardır:

- Görevlere (tasks) çok yüksek bir öncelik atanmış olsa bile, bu görevler yalnızca donanım tarafından hiçbir kesmeye hizmet verilmediğinde çalışacaktır.
- ISR'ler bir görevin hem başlama zamanını hem de yürütme süresini bozabilir (buna 'seğirme' - 'jitter' eklenebilir).
- FreeRTOS'un çalıştığı mimariye bağlı olarak, bir ISR yürütülürken herhangi bir yeni kesmenin veya en azından bir grup yeni kesmenin kabul edilmesi mümkün olmayabilir.
- Uygulama yazarının, değişkenler, çevre birimleri ve bellek arabellekleri (memory buffers) gibi kaynaklara bir görev ve bir ISR tarafından aynı anda (at the same time) erişilmesinin sonuçlarını dikkate alması ve bunlara karşı önlem alması gerekir.
- Bazı FreeRTOS portları kesmelerin yuvalanmasına (nest) izin verir, ancak kesme yuvalaması karmaşıklığı artırabilir ve öngörülebilirliği azaltabilir. Bir kesme ne kadar kısaysa, yuvalanma olasılığı da o kadar düşüktür.

Bir kesme servis rutini kesmenin nedenini kaydetmeli ve kesmeyi temizlemelidir (clear the interrupt). Kesmenin gerektirdiği diğer işlemler genellikle bir görevde gerçekleştirilebilir ve böylece kesme servis rutininin pratik olduğu kadar hızlı bir şekilde çıkmasına (exit) olanak tanınır. Buna 'ertelenmiş kesme işleme' (deferred interrupt processing) denir, çünkü kesmenin gerektirdiği işlem ISR'den bir göreve 'ertelenir'.

Kesme işleminin bir göreve ertelenmesi, uygulama yazarının işlemi uygulamadaki diğer görevlere göre önceliklendirmesine ve tüm FreeRTOS API işlevlerini kullanmasına da olanak tanır.

Kesme işleminin ertelendiği görevin önceliği diğer tüm görevlerin önceliğinin üzerindeyse, işlem tıpkı ISR'nin kendisinde gerçekleştirilmiş gibi anında gerçekleştirilecektir. Bu senaryo, Görev 1'in (Task 1) normal bir uygulama görevi olduğu ve Görev 2'nin (Task 2) kesme işleminin ertelendiği görev olduğu Şekil 7.1'de gösterilmektedir.



Şekil 7.1 Yüksek öncelikli bir görevde kesme işleminin tamamlanması

Şekil 7.1'de, kesme işlemi **t2** zamanında başlar ve etkili bir şekilde **t4** zamanında sona erer, ancak ISR'de yalnızca **t2** ve **t3** zamanları arasındaki periyot harcanır. Ertelenmiş kesme işlemi kullanılmamış olsaydı, **t2** ve **t4** zamanları arasındaki periyodun tamamı ISR'de harcanacaktı.

Bir kesmenin gerektirdiği tüm işlemlerin ne zaman ISR'de yapılmasının en iyisi olduğu ve işlemin bir kısmının ne zaman bir göreve ertelenmesinin en iyisi olduğu konusunda kesin bir kural yoktur. İşlemi bir göreve ertelemek şu durumlarda en kullanışlısıdır:

- Kesmenin gerektirdiği işlem önemsiz değilse. Örneğin, kesme yalnızca analogdan dijitale dönüştürmenin (analog to digital conversion) sonucunu depoluyorsa, bunun en iyi ISR içinde gerçekleştirileceği neredeyse kesindir, ancak dönüştürme sonucunun bir yazılım filtresinden geçirilmesi de gerekiyorsa, filtreyi bir görevde çalıştırmak en iyisi olabilir.
- Bir konsola yazmak veya bellek tahsis etmek (allocate memory) gibi bir ISR içinde gerçekleştirilemeyecek bir eylemi kesme işleminin gerçekleştirmesi uygun olduğunda.
- Kesme işlemi deterministik değilse — yani işlemin ne kadar süreceği önceden bilinmiyorsa.

Aşağıdaki bölümler, ertelenmiş kesme işlemeyi uygulamak için kullanılabilecek FreeRTOS özellikleri de dahil olmak üzere, bu bölümde şimdiye kadar tanıtılan kavramları açıklamakta ve göstermektedir.

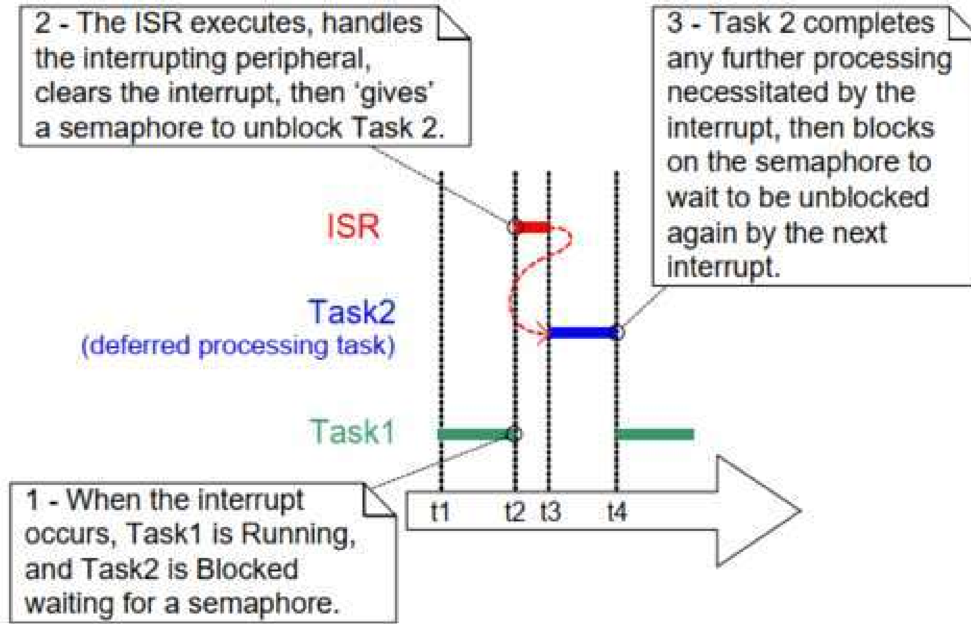
7.4 Senkronizasyon (Eşzamanlama) İçin Kullanılan İkili Semaforlar (Binary Semaphores)

İkili Semafor API'sinin kesme güvenli (interrupt safe) sürümü, belirli bir kesme her meydana geldiğinde bir görevin engelini kaldırmak (unblock) için kullanılabilir ve böylece görev ile kesme etkin bir şekilde senkronize edilebilir (eşzamanlanabilir). Bu, kesme olayı işleminin (interrupt event processing) büyük bir kısmının senkronize edilmiş görev içinde uygulanmasına olanak tanırken, doğrudan ISR'de yalnızca çok hızlı ve kısa bir kısım kalır. Önceki bölümde açıklandığı gibi, ikili semafor kesme işlemini bir göreve 'ertelemek' için kullanılır.

(Not: Bir görevi kesmeden itibaren doğrudan göreve bildirim - direct to task notification - kullanarak unblock yapmak, ikili semafor kullanmaktan daha verimlidir. Doğrudan göreve bildirimler, Bölüm 10'a kadar işlenmemiştir.)

Şekil 7.1'de daha önce gösterildiği gibi, kesme işlemi özellikle zaman açısından kritikse (time critical), ertelenen işlem görevinin (deferred processing task) önceliği, görevin her zaman sistemdeki diğer görevleri engellemesini (pre-empt) sağlayacak şekilde ayarlanabilir. Daha sonra ISR, `portYIELD_FROM_ISR()` işlevine bir çağrı içerecek şekilde uygulanabilir ve böylece ISR'nin doğrudan kesme işleminin ertelendiği göreve dönmesi sağlanır. Bu, tüm olay işlemenin (event processing) sanki tamamı ISR'nin

içinde uygulanmış gibi zaman içinde bitişik (contiguous - kesintisiz olarak) çalışmasını sağlama etkisine sahiptir. Şekil 7.2, Şekil 7.1'de gösterilen senaryoyu tekrarlar, ancak metin, ertelenen işleme görevinin (deferred processing task) yürütülmesinin bir semafor kullanılarak nasıl kontrol edilebileceğini açıklayacak şekilde güncellenmiştir.



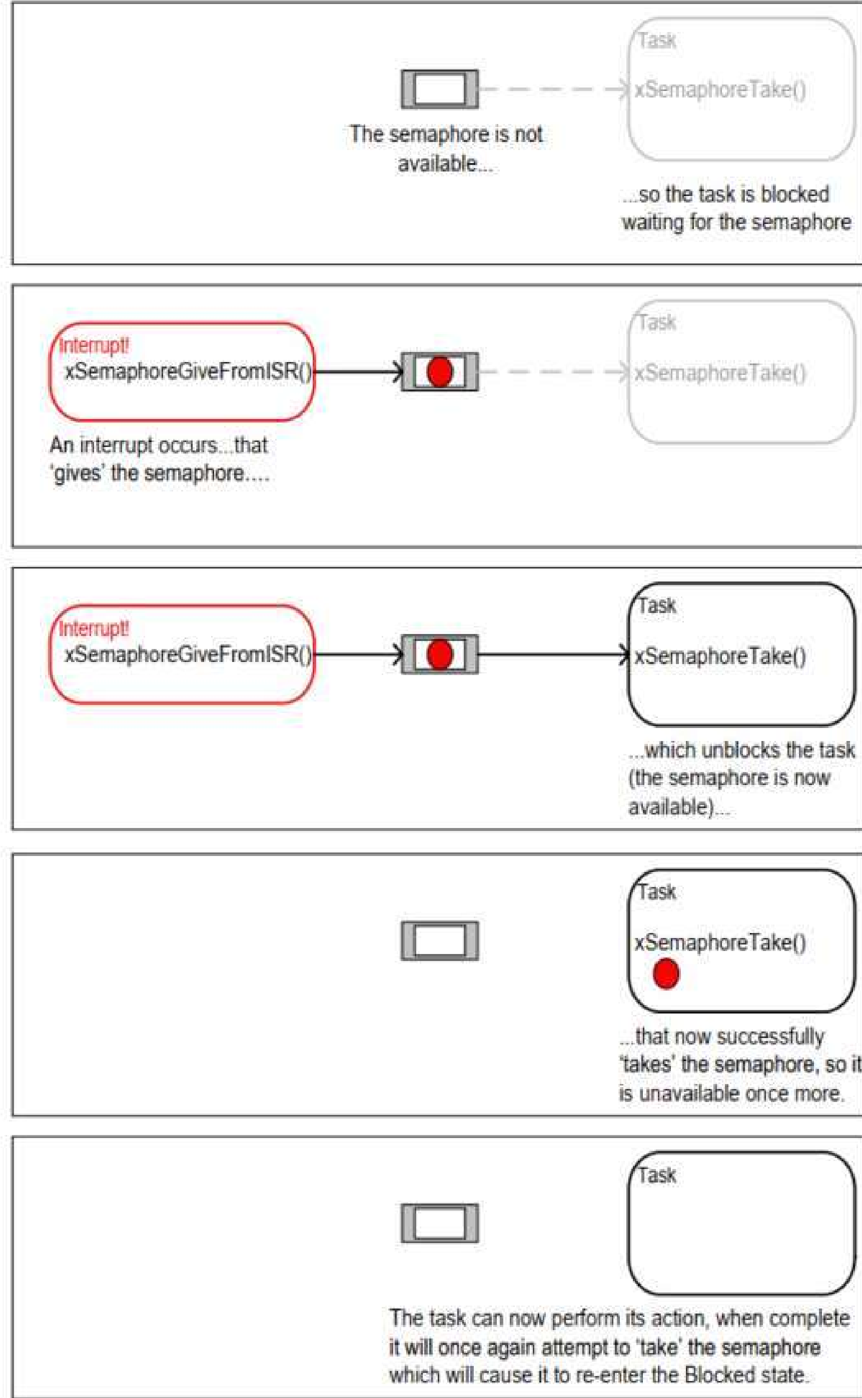
Şekil 7.2 Ertelenmiş kesme işlemeyi uygulamak için bir ikili semafor kullanma

Ertelenen işlem görevi, olayın gerçekleşmesini beklemek üzere Engellenmiş durumuna (Blocked state) girmek için bir semafora yönelik engelleyici 'alma' (blocking 'take') çağrısını kullanır. Olay gerçekleştiğinde, ISR gerekli olay işlemenin (event processing) ilerleyebilmesi için görevin engelini kaldırmak (unblock) üzere aynı semafor üzerinde bir 'verme' (give) işlemi kullanır.

'Semafor almak' ve 'semafor vermek' kullanım senaryolarına bağlı olarak farklı anlamlara gelen kavramlardır. Bu kesme senkronizasyon senaryosunda (interrupt synchronization scenario), ikili semafor kavramsal olarak uzunluğu bir olan bir kuyruk olarak düşünülebilir. Kuyruk herhangi bir zamanda en fazla bir öğe içerebilir, bu nedenle her zaman ya boştur ya da doludur (dolayısıyla ikilidir - binary). `xSemaphoreTake()` öğesini çağırarak, kesme işleminin ertelendiği görev, etkili bir şekilde kuyruktan bir blok süresiyle (block time) okuma yapmaya çalışır ve kuyruk boşsa görevin Engellenmiş durumuna girmesine neden olur. Olay gerçekleştiğinde ISR, kuyruğa bir belirteç (token - semafor) yerleştirmek için `xSemaphoreGiveFromISR()` işlevini kullanır ve kuyruğu dolu hale getirir. Bu, görevin Engellenmiş durumundan çıkmasına ve belirteci kaldırmasına neden olarak kuyruğu bir kez daha boş bırakır. Görev işlemini tamamladığında, kuyruktan okumayı bir kez daha dener ve kuyruğu boş bularak bir sonraki olayı beklemek üzere yeniden Engellenmiş durumuna girer. Bu dizi Şekil 7.3'te gösterilmektedir.

Şekil 7.3, ilk olarak (öncesinde) onu 'almamış' olmasına rağmen kesmenin semaforu 'verdiğini' ve görevin semaforu 'aldığını' ancak onu asla geri vermediğini

göstermektedir. Senaryonun kavramsal olarak bir kuyruğa yazmaya ve kuyruktan okumaya benzer olarak tanımlanmasının nedeni budur. Bu durum çoğu zaman kafa karışıklığına neden olur, çünkü bir semafor alan görevin her zaman onu geri vermesi gerektiği diğer semafor kullanım senaryolarıyla aynı kurallara uymaz - örneğin Bölüm 8, Kaynak Yönetimi'nde açıklanan senaryolar gibi.



Şekil 7.3 Bir görevi bir kesmeyle eşzamanlamak (senkronize etmek) için ikili semafor kullanımı

7.4.1 xSemaphoreCreateBinary() API İşlevi

FreeRTOS ayrıca derleme zamanında (compile time) statik olarak ikili bir semafor oluşturmak için gereken belleği tahsis eden `xSemaphoreCreateBinaryStatic()` işlevini de içerir. Çeşitli FreeRTOS semafor türlerinin hepsine yönelik tanıtıcılar (handles), `SemaphoreHandle_t` türündeki bir değişkende saklanır.

Bir semafor kullanılmadan önce açıkça (explicitly) oluşturulmalıdır. İkili bir semafor oluşturmak için `xSemaphoreCreateBinary()` API işlevini kullanın.

Not: Bazı Semaphore API işlevleri aslında işlev değil makrolardır. Basitlik sağlamak için bu kitap boyunca hepsinden işlev olarak bahsedilecektir.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Liste 7.3 `xSemaphoreCreateBinary()` API işlev prototipi

`xSemaphoreCreateBinary()` Dönüş Değeri:

- `NULL` döndürülürse, FreeRTOS'un semafor veri yapılarını tahsis etmesi için yeterli yığın belleği (heap memory) bulunmadığından semafor oluşturulamaz.
- `NULL` olmayan bir değer döndürülürse bu, semaforun başarıyla oluşturulduğunu gösterir. Döndürülen değer, oluşturulan semaforun tanıtıcısı (handle) olarak kaydedilmelidir.

7.4.2 xSemaphoreTake() API İşlevi

Bir semaforu 'almak' (taking), semaforu 'elde etmek' (obtain) veya 'teslim almak' (receive) anlamına gelir. Semafor yalnızca mevcut (available) ise alınabilir.

Yinelemeli (recursive) muteksler hariç olmak üzere, çeşitli FreeRTOS semafor türlerinin tümü `xSemaphoreTake()` işlevi kullanılarak 'alınabilir'.

`xSemaphoreTake()`, bir kesme servis rutininden (ISR) kullanılmamalıdır.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait
);
```

Liste 7.4 `xSemaphoreTake()` API işlev prototipi

`xSemaphoreTake()` parametreleri ve dönüş değeri:

- **xSemaphore:** 'Alınan' semafor. Bir semafora `SemaphoreHandle_t` türünde bir değişken referans verir. Kullanılmadan önce açıkça oluşturulmalıdır.
- **`xTicksToWait`:** Zaten mevcut değilse, görevin semaforu beklemek için Engellenmiş durumunda kalması gereken maksimum süre. `xTicksToWait`

sıfırsa, semafor mevcut olmadığında `xSemaphoreTake()` işlevi anında geri dönecektir. Blok süresi (block time) tick periyotları cinsinden belirtilir. `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak, `FreeRTOSConfig.h` içinde `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, görevin süresiz olarak (zaman aşımına uğramadan) beklemesine neden olacaktır.

- **Dönüş değeri:** İki olası dönüş değeri vardır:
  - **`pdPASS`**: Yalnızca `xSemaphoreTake()` çağrısı semaforu elde etmede başarılı olduysa döndürülür. Bir blok süresi belirtildiyse, semafor hemen kullanılamadığında çağıran görev Engellenmiş durumuna yerleştirilmiş olabilir, ancak blok süresi dolmadan önce semafor kullanılabilir hale gelmiştir.
  - **`pdFALSE`**: Semafor mevcut değildir. Bir blok süresi belirtilmişse çağıran görev, semaforun uygun olmasını beklemek üzere Engellenmiş durumuna yerleştirilecek, ancak bu gerçekleşmeden önce blok süresi dolacaktır.

7.4.3 xSemaphoreGiveFromISR() API İşlevi

İkili (binary) ve sayıcı (counting) semaforlar `xSemaphoreGiveFromISR()` işlevi kullanılarak 'verilebilir' (given). (Sayıcı semaforlar bu kitabın ilerleyen bir bölümünde açıklanmıştır.)

`xSemaphoreGiveFromISR()`, `xSemaphoreGive()` işlevinin kesme güvenli sürümüdür, bu nedenle bu bölümün başında açıklanan `pxHigherPriorityTaskWoken` parametresine sahiptir.

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,

                                  BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 7.5 `xSemaphoreGiveFromISR()` API işlev prototipi

`xSemaphoreGiveFromISR()` parametreleri ve dönüş değeri:

- **xSemaphore:** 'Verilmekte' olan semafor.
- **`pxHigherPriorityTaskWoken`:** Tek bir semaforun üzerinde semaforun uygun olmasını bekleyen bir veya daha fazla görev (task) olması mümkündür. `xSemaphoreGiveFromISR()` işlevinin çağrılması semaforu uygun hale getirebilir ve böylece semaforu bekleyen bir görevin Engellenmiş durumundan çıkmasına neden olabilir. Eğer `xSemaphoreGiveFromISR()` çağrısı bir görevin Engellenmiş durumundan çıkmasına neden olursa ve engeli kaldırılan görevin (unblocked task) önceliği o anda çalışmakta olan görevin (kesintiye uğratılan görevin) önceliğinden yüksekse, `xSemaphoreGiveFromISR()` dahili (internally) olarak `*pxHigherPriorityTaskWoken` değerini `pdTRUE` olarak ayarlayacaktır. Bu gerçekleşirse, normalde kesmeden çıkılmadan önce bir bağlam geçişi (context switch) gerçekleştirilmelidir.
- **Dönüş değeri:** İki olası dönüş değeri vardır:

- pdPASS: `xSemaphoreGiveFromISR()` çağrısı yalnızca başarılı olursa döndürülür.
- pdFAIL: Bir semafor zaten mevcutsa verilemez ve `xSemaphoreGiveFromISR()` işlevi `pdFAIL` döndürür.

Örnek 7.1 Bir görevi bir kesmeyle senkronize etmek için ikili bir semafor kullanma

Bu örnek, bir kesme servis rutininden (ISR) bir görevin engelini kaldırmak ve böylece görevi kesmeyle senkronize etmek için bir ikili semafor kullanır.

Her 500 milisaniyede bir yazılım kesmesi (software interrupt) üretmek için basit bir periyodik görev kullanılır. Bazı hedef ortamlarda gerçek bir kesmeye bağlanmanın karmaşıklığı nedeniyle kolaylık sağlamak amacıyla bir yazılım kesmesi kullanılmıştır.

Liste 7.6 periyodik görevin uygulanmasını (implementation) göstermektedir. Görevin, kesme üretilmeden önce ve sonra bir dize (string) yazdırdığına dikkat edin. Bu, örnek çalıştırıldığında üretilen çıktıda yürütme dizisinin (sequence of execution) gözlemlenmesini sağlar.

```
/* Bu örnekte kullanılan yazılım kesmesinin numarası. Gösterilen kod Windows
projesinden

   alınmıştır; burada 0 ile 2 arasındaki sayılar FreeRTOS Windows portunun kendisi

   tarafından kullanılır, bu nedenle 3 uygulamanın kullanabileceği ilk sayıdır. */

#define mainINTERRUPT_NUMBER 3


static void vPeriodicTask( void *pvParameters )

{

    const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );


    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü içinde uygulanır. */

    for( ;; )

    {

        /* Yazılım kesmesini tekrar üretme zamanı gelene kadar engellen. */

        vTaskDelay( xDelay500ms );


        /* Yürütme dizisi çıktıdan belirgin olsun diye kesme üretilmeden

           önce ve sonra bir mesaj yazdırarak kesmeyi üret.
```

```
          Yazılım kesmesi üretmek için kullanılan sözdizimi (syntax)

          kullanılan FreeRTOS portuna (uyarlamasına) bağlıdır. Aşağıda kullanılan

          sözdizimi yalnızca bu tür kesmelerin simüle edildiği FreeRTOS

          Windows portunda kullanılabilir. */

        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );

        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );

    }

}
```

Liste 7.6 Örnek 7.1'de periyodik olarak bir yazılım kesmesi üreten görevin uygulanması

Liste 7.7, kesme işleminin ertelendiği (interrupt processing is deferred) görevin (yani, bir ikili semafor kullanımı yoluyla yazılım kesmesiyle senkronize edilen görevin) uygulanmasını göstermektedir. Yine, görevin her yinelemesinde (iteration) bir dize (string) yazdırılır, böylece görev ve kesmenin hangi sırayla yürütüldüğü, örnek çalıştırıldığında üretilen çıktıdan açıkça görülür.

Liste 7.7'de gösterilen kodun kesmelerin yazılım tarafından üretildiği Örnek 7.1 için yeterli olduğu, ancak kesmelerin donanım çevre birimleri (hardware peripherals) tarafından üretildiği senaryolar için yeterli olmadığı unutulmamalıdır. Sonraki bir alt bölümde, kodun yapısının donanım tarafından üretilen kesmelerle kullanıma uygun hale getirilmesi için nasıl değiştirilmesi gerektiği açıklanmaktadır.

```
static void vHandlerTask( void *pvParameters )

{

    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü içinde uygulanır. */

    for( ;; )

    {

        /* Olayı beklemek için semaforu kullanın. Semafor, çizgeleyici (scheduler)

           başlatılmadan önce, yani bu görev ilk kez çalışmadan önce
oluşturulmuştu.

           Görev süresiz olarak engellenir, bu da bu işlev çağrısının ancak

           semafor başarıyla elde edildikten sonra geri döneceği anlamına gelir

           - bu nedenle xSemaphoreTake() tarafından döndürülen değeri kontrol
```

```
          etmeye gerek yoktur. */

        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );


        /* Buraya gelmek için olayın gerçekleşmiş olması gerekir. Olayı

           işleyin (bu durumda, sadece bir mesaj yazdırın). */

        vPrintString( "Handler task - Processing event.\r\n" );

    }

}
```

Liste 7.7 Örnek 7.1'de kesme işleminin ertelendiği görevin (kesmeyle senkronize olan görevin) uygulanması

Liste 7.8, ISR'yi (Kesme Servis Rutini) göstermektedir. Bu, kesme işleminin ertelendiği görevin engelini kaldırmak için semaforu 'vermekten' (give) başka çok az şey yapar.

`xHigherPriorityTaskWoken` değişkeninin nasıl kullanıldığına dikkat edin. `xSemaphoreGiveFromISR()` işlevi çağrılmadan önce `pdFALSE` olarak ayarlanır, ardından `portYIELD_FROM_ISR()` çağrıldığında parametre olarak kullanılır. Eğer `xHigherPriorityTaskWoken` `pdTRUE` değerine eşitse, `portYIELD_FROM_ISR()` makrosu içinde bir bağlam geçişi (context switch) talep edilecektir.

ISR'nin prototipi ve bağlam geçişini zorlamak için çağrılan makro, FreeRTOS Windows portu için doğrudur ve diğer FreeRTOS portları için farklı olabilir. Kullanmakta olduğunuz port için gereken sözdizimini (syntax) bulmak amacıyla FreeRTOS.org web sitesindeki porta özgü belgelere ve FreeRTOS indirmesinde sağlanan örneklere bakın.

FreeRTOS'un çalıştığı çoğu mimarinin aksine, FreeRTOS Windows portu bir ISR'nin bir değer döndürmesini gerektirir. Windows portu ile birlikte sağlanan `portYIELD_FROM_ISR()` makrosunun uygulanması dönüş (return) ifadesini (statement) içerir, bu nedenle Liste 7.8 açıkça bir değer döndürüldüğünü göstermez.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;


    /* xHigherPriorityTaskWoken parametresi, eğer bir bağlam geçişi

       gerekirse kesme güvenli (interrupt safe) API işlevi içinde pdTRUE olarak

       ayarlanacağından pdFALSE olarak başlatılmalıdır. */

    xHigherPriorityTaskWoken = pdFALSE;
```

```
    /* xHigherPriorityTaskWoken değişkeninin adresini kesme güvenli API işlevinin
       pxHigherPriorityTaskWoken parametresi olarak geçirerek görevin engelini
       kaldırmak için semaforu 'Verin' (Give). */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* xHigherPriorityTaskWoken değerini portYIELD_FROM_ISR() işlevine geçirin.
       xHigherPriorityTaskWoken, xSemaphoreGiveFromISR() içinde pdTRUE olarak
       ayarlandıysa portYIELD_FROM_ISR() çağrısı bir bağlam geçişi talep
       edecektir. xHigherPriorityTaskWoken hala pdFALSE ise,
       portYIELD_FROM_ISR() çağrısının hiçbir etkisi olmayacaktır. Çoğu FreeRTOS
       portunun (uyarlamasının) aksine, Windows portu ISR'nin bir değer
       döndürmesini gerektirir - return ifadesi portYIELD_FROM_ISR()
       makrosunun Windows sürümünün içindedir. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Liste 7.8 Örnek 7.1'de kullanılan yazılım kesmesi için ISR

`main()` işlevi ikili semaforu oluşturur, görevleri oluşturur, kesme işleyiciyi (interrupt handler) kurar ve çizgeleyiciyi başlatır. Uygulama Liste 7.9'da gösterilmektedir.

Bir kesme işleyicisini kurmak (install) için çağrılan işlevin sözdizimi FreeRTOS Windows portuna özgüdür ve diğer FreeRTOS portları için farklı olabilir. Kullanmakta olduğunuz port için gereken sözdizimini bulmak için FreeRTOS.org web sitesindeki porta özgü belgelere ve FreeRTOS indirmesinde sağlanan örneklere bakın.

```
int main( void )
{
    /* Bir semafor kullanılmadan önce açıkça oluşturulmalıdır. Bu örnekte
       ikili (binary) bir semafor oluşturulmaktadır. */
    xBinarySemaphore = xSemaphoreCreateBinary();

    /* Semaforun başarıyla oluşturulduğunu kontrol edin. */
```

```
    if( xBinarySemaphore != NULL )

    {

        /* Kesme işleminin ertelendiği görev olan 'işleyici' ('handler')

           görevini oluşturun. Bu görev, kesme ile senkronize edilecek

           görevdir. İşleyici görev, kesmeden çıkıldıktan (exits) hemen sonra

           çalışmasını sağlamak için yüksek bir öncelikle oluşturulur.

           Bu durumda 3 önceliği seçilmiştir. */

        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );


        /* Periyodik olarak yazılım kesmesi üretecek olan görevi oluşturun.

           Bu görev, işleyici görevinin Engellenmiş durumundan her çıkışında
(exits

           the Blocked state) işleyici görev tarafından engellenmesini (pre-
empted)

           sağlamak için işleyici görevinden daha düşük bir öncelikte oluşturulur.
*/

        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );


        /* Yazılım kesmesi için işleyiciyi kurun. Bunu yapmak için gereken

           sözdizimi kullanılan FreeRTOS portuna bağlıdır. Burada gösterilen

           sözdizimi yalnızca bu tür kesmelerin simüle edildiği FreeRTOS

           windows portunda kullanılabilir. */

        vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler
);


        /* Oluşturulan görevlerin çalışmaya başlaması için çizgeleyiciyi başlatın.
*/

        vTaskStartScheduler();

    }


    /* Normalde aşağıdaki satıra asla ulaşılmamalıdır. */

    for( ;; );
```

```
}
```

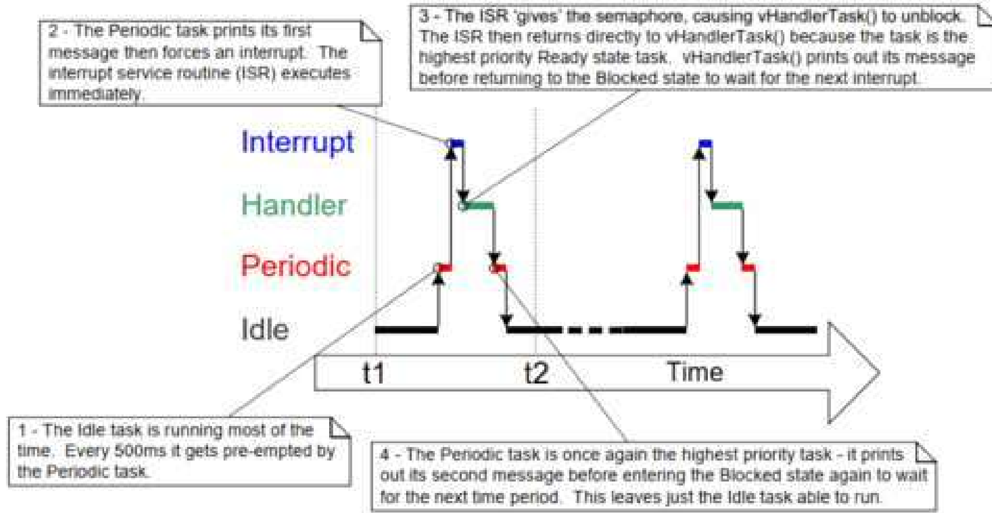
Liste 7.9 Örnek 7.1 için `main()` işlevinin uygulanması

Örnek 7.1, Şekil 7.4'te gösterilen çıktıyı üretir. Beklendiği gibi, kesme üretilir üretilmez `vHandlerTask()` Çalışıyor durumuna (Running state) girer, böylece görevden (task) gelen çıktı periyodik görev tarafından üretilen çıktıyı böler (splits).

Daha fazla açıklama Şekil 7.5'te sağlanmaktadır.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
```

Şekil 7.4 Örnek 7.1 yürütüldüğünde üretilen çıktı



Şekil 7.5 Örnek 7.1 yürütüldüğünde yürütme dizisi (sequence of execution)

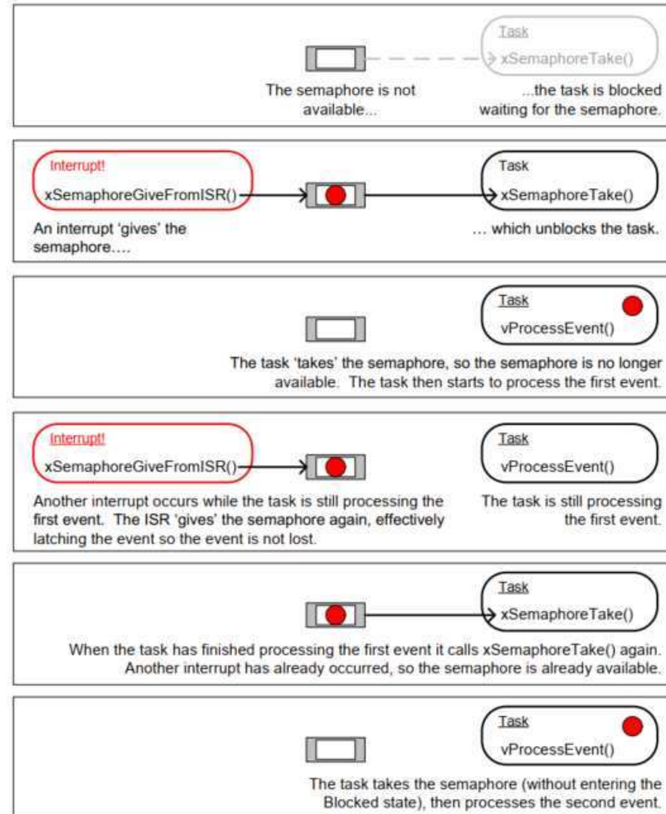
7.4.4 Örnek 7.1'de Kullanılan Görevin Uygulamasının Geliştirilmesi

Örnek 7.1, bir görevi bir kesmeyle senkronize etmek için ikili bir semafor kullanmıştır. Yürütme dizisi aşağıdaki gibiydi:

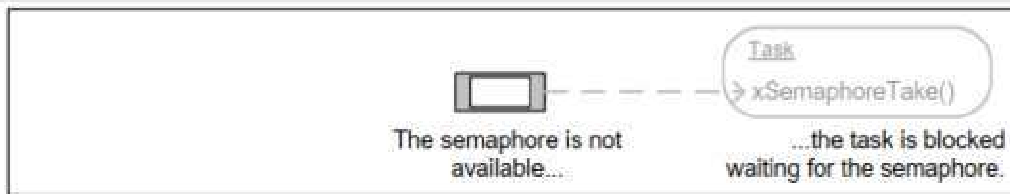
1. Kesme (interrupt) meydana geldi.
2. ISR (Kesme Servis Rutini) çalıştı ve görevin engelini kaldırmak için semaforu 'verdi' (gave).
3. Görev ISR'den hemen sonra çalıştı ve semaforu 'aldı' (took).
4. Görev olayı işledi, ardından semaforu tekrar 'almaya' çalıştı—semafor henüz mevcut olmadığı için Engellenmiş (Blocked) durumuna girdi (başka bir kesme henüz meydana gelmemişti).

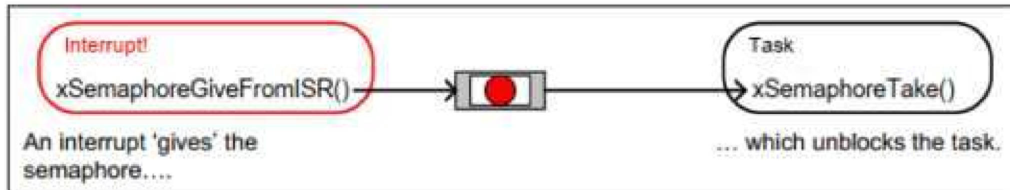
Örnek 7.1'de kullanılan görevin yapısı yalnızca kesmeler nispeten düşük bir frekansta (frequency) meydana gelirse yeterlidir. Nedenini anlamak için, görev ilk kesmeyi işlemeyi tamamlamadan önce ikinci ve ardından üçüncü bir kesme meydana gelmiş olsaydı ne olacağını düşünün:

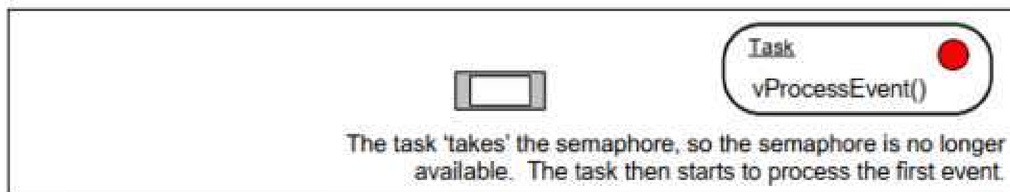
- İkinci ISR yürütüldüğünde, semafor boş olacağından ISR semaforu verir ve görev ilk olayı işlemeyi bitirdikten hemen sonra ikinci olayı işler. Bu senaryo Şekil 7.6'da gösterilmektedir.
- Üçüncü ISR yürütüldüğünde semafor zaten mevcut olurdu ve bu durum ISR'nin semaforu tekrar vermesini engellerdi, bu nedenle görev üçüncü olayın meydana geldiğini bilmezdi. Bu senaryo Şekil 7.7'de gösterilmektedir.

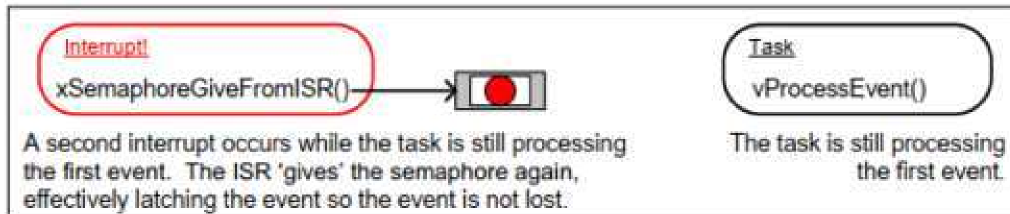


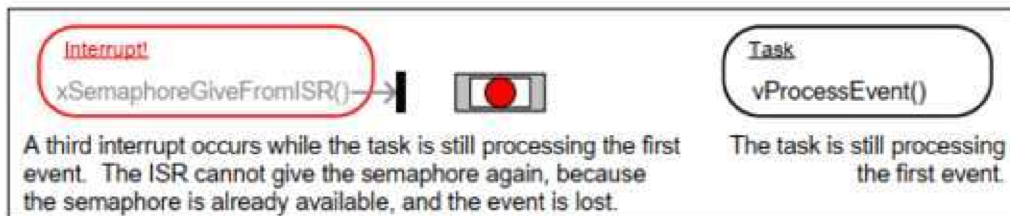
Şekil 7.6 Görev ilk olayı işlemeyi bitirmeden önce bir kesme daha meydana geldiğindeki senaryo

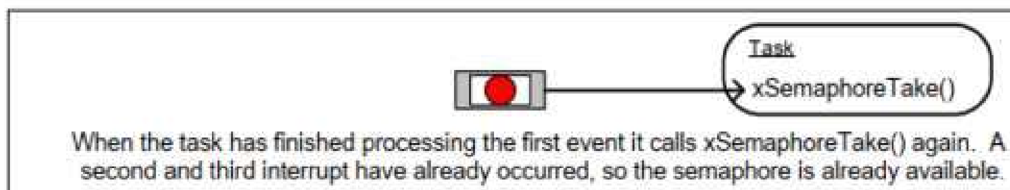

Task
xSemaphoreTake()
The semaphore is not available...
...the task is blocked waiting for the semaphore.

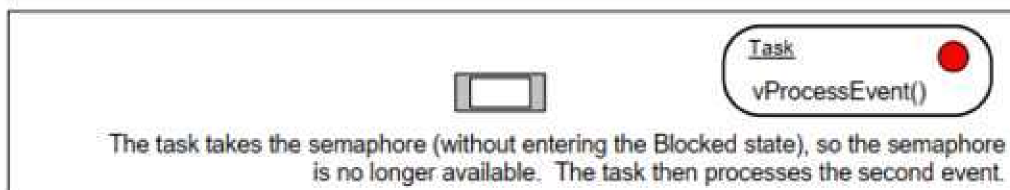

Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt 'gives' the semaphore....
... which unblocks the task.

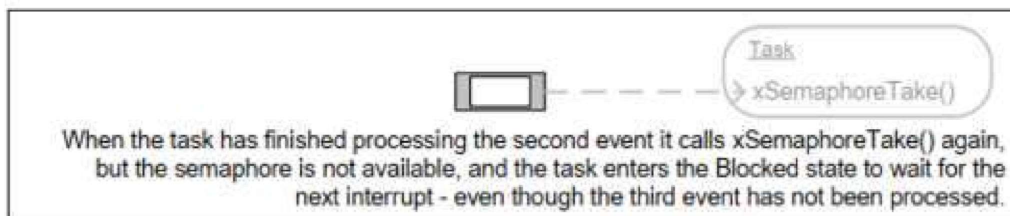

Task
vProcessEvent()
The task 'takes' the semaphore, so the semaphore is no longer available. The task then starts to process the first event.


Interrupt!
xSemaphoreGiveFromISR()
Task
vProcessEvent()
A second interrupt occurs while the task is still processing the first event. The ISR 'gives' the semaphore again, effectively latching the event so the event is not lost.
The task is still processing the first event.


Interrupt!
xSemaphoreGiveFromISR()
Task
vProcessEvent()
A third interrupt occurs while the task is still processing the first event. The ISR cannot give the semaphore again, because the semaphore is already available, and the event is lost.
The task is still processing the first event.


Task
xSemaphoreTake()
When the task has finished processing the first event it calls xSemaphoreTake() again. A second and third interrupt have already occurred, so the semaphore is already available.


Task
vProcessEvent()
The task takes the semaphore (without entering the Blocked state), so the semaphore is no longer available. The task then processes the second event.


Task
xSemaphoreTake()
When the task has finished processing the second event it calls xSemaphoreTake() again, but the semaphore is not available, and the task enters the Blocked state to wait for the next interrupt - even though the third event has not been processed.

Şekil 7.7 Görev ilk olayı işlemeyi bitirmeden önce iki kesme meydana geldiğindeki senaryo

Örnek 7.1'de kullanılan ve Liste 7.7'de gösterilen ertelenmiş kesme işleme görevi (deferred interrupt handling task), her `xSemaphoreTake()` çağrısı arasında yalnızca bir olayı işleyecek şekilde yapılandırılmıştır. Olayları üreten kesmeler yazılım tarafından tetiklendiği (triggered by software) ve öngörülebilir bir zamanda (predictable time) meydana geldiği için bu Örnek 7.1 için yeterliydi. Gerçek uygulamalarda kesmeler donanım tarafından üretilir ve öngörülemeyen zamanlarda meydana gelir. Bu nedenle, bir kesmenin gözden kaçma (missed) olasılığını en aza indirmek için, ertelenmiş kesme işleme görevi, her `xSemaphoreTake()` çağrısı arasında halihazırda mevcut olan tüm olayları işleyecek şekilde yapılandırılmalıdır.

(Not: Alternatif olarak, olayları saymak için bir sayıcı semafor - counting semaphore - veya doğrudan göreve bildirim - direct to task notification - kullanılabilir. Sayıcı semaforlar bir sonraki bölümde açıklanmıştır. Doğrudan göreve bildirimler ise hem çalışma zamanı hem de RAM kullanımı açısından en verimli yöntem oldukları için tercih edilen yöntemdir ve Bölüm 9'da açıklanmıştır.)

Örnek 7.1'de kullanılan ertelenmiş kesme işleme görevinin bir başka zayıflığı daha vardı; `xSemaphoreTake()` işlevini çağırdığında bir zaman aşımı (timeout) kullanmıyordu. Bunun yerine görev, `xSemaphoreTake()` işlevinin `xTicksToWait` parametresi olarak `portMAX_DELAY` değerini geçiyordu, bu da görevin semaforun uygun olmasını süresiz olarak (zaman aşımı olmadan) beklemesine neden olur.

Belirsiz zaman aşımları (indefinite timeouts), kullanımının örneğin yapısını basitleştirmesi ve dolayısıyla örneğin anlaşılmasını kolaylaştırması nedeniyle örnek kodlarda sıklıkla kullanılır. Ancak, bir hatadan (error) kurtulmayı (recover) zorlaştırdıkları için, belirsiz zaman aşımları gerçek uygulamalarda genellikle kötü bir uygulamadır (bad practice). Örnek olarak, bir görevin bir kesmenin semafor vermesini beklediği, ancak donanımdaki bir hata durumunun kesmenin üretilmesini engellediği senaryoyu düşünün:

- Görev zaman aşımı olmadan bekliyorsa, hata durumunu bilmeyecek ve sonsuza kadar bekleyecektir.
- Görev bir zaman aşımı ile bekliyorsa, zaman aşımı süresi dolduğunda `xSemaphoreTake()` `pdFAIL` döndürecek ve görev bir sonraki yürütülüşünde hatayı tespit edip temizleyebilecektir. Bu senaryo Liste 7.10'da da gösterilmektedir.

```
static void vUARTReceiveHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime, iki kesme arasında beklenen maksimum süreyi tutar.
*/

    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );



    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü içinde uygulanır. */

    for( ;; )
```

```
    {
        /* Semafor UART'ın alma (Rx) kesmesi tarafından 'verilir' (given).
           Bir sonraki kesme için maksimum xMaxExpectedBlockTime tick bekleyin. */
        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS )
        {
            /* Semafor elde edildi. Tekrar xSemaphoreTake() öğesini
               çağırmadan önce bekleyen (pending) Rx olaylarının TÜMÜNÜ işleyin.
               Her bir Rx olayı UART'ın alma FIFO'suna (arabelleğine) bir karakter
               yerleştirmiş olacaktır ve UART_RxCount() işlevinin FIFO'daki
               karakter sayısını döndürdüğü varsayılmaktadır. */
            while( UART_RxCount() > 0 )
            {
                /* UART_ProcessNextRxEvent() işlevinin bir Rx karakterini
                   işlediği ve FIFO'daki karakter sayısını 1 azalttığı
                   varsayılmaktadır. */
                UART_ProcessNextRxEvent();
            }

            /* Bekleyen (pending) başka Rx olayı yoktur (FIFO'da başka karakter
               kalmamıştır), bu nedenle geriye dönün (loop back) ve bir sonraki
               kesmeyi beklemek için xSemaphoreTake() işlevini çağırın. Koddaki
               bu nokta ile xSemaphoreTake() çağrısı arasında meydana gelen tüm
               kesmeler semafora kaydedilecek (latched) ve böylece
kaybolmayacaktır. */
        }
        else
        {
            /* Beklenen süre içinde bir olay alınmadı. UART'ın daha fazla
               kesme üretmesini engelleyebilecek herhangi bir hata durumu olup
               olmadığını kontrol edin ve gerekirse temizleyin (clear). */
```

```
            UART_ClearErrors();

        }

    }

}
```

Liste 7.10 Örnek olarak bir UART alma işleyicisi (receive handler) kullanılarak, ertelenmiş bir kesme işleme görevinin önerilen yapısı

7.5 Sayıcı Semaforlar (Counting Semaphores)

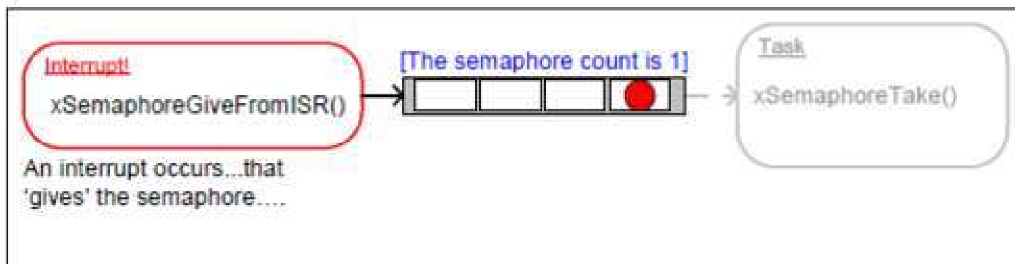
İkili semaforlar nasıl uzunluğu bir olan kuyruklar olarak düşünülebilirse, sayıcı semaforlar da uzunluğu birden fazla olan kuyruklar olarak düşünülebilir. Görevler (tasks) kuyrukta depolanan verilerle ilgilenmez, sadece kuyruktaki öğelerin sayısıyla ilgilenir. Sayıcı semaforların kullanılabilmesi için `FreeRTOSConfig.h` içinde `configUSE_COUNTING_SEMAPHORES` öğesinin 1 olarak ayarlanması gerekir.

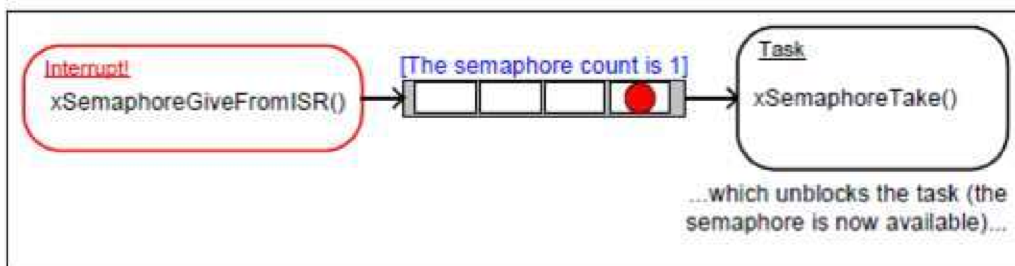
Sayıcı bir semafor her 'verildiğinde' (given), kuyruğundaki başka bir alan kullanılır. Kuyruktaki öğelerin sayısı, semaforun 'sayım' (count) değeridir.

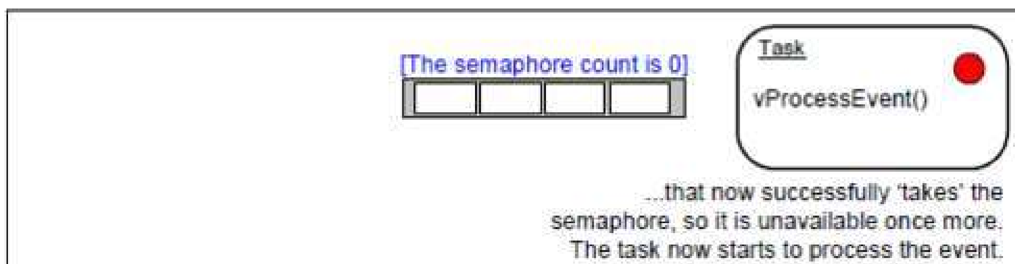
Sayıcı semaforlar tipik olarak iki şey için kullanılır:

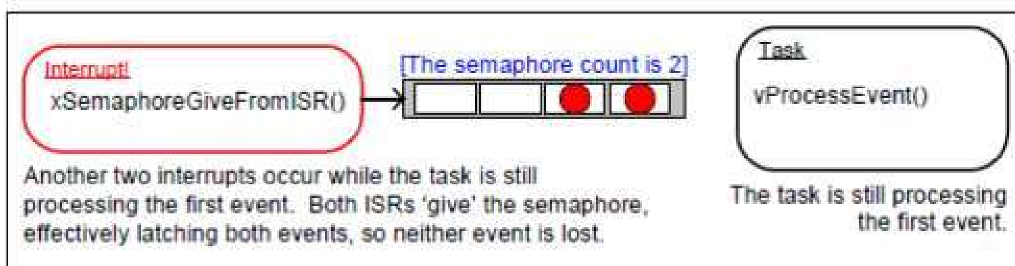
1. **Olayları sayma (Counting events):** Bu senaryoda, bir olay işleyicisi (event handler) her olay gerçekleştiğinde bir semafor 'verecek' (give) ve bu da her 'verme' işleminde semaforun sayım (count) değerinin artmasına (incremented) neden olacaktır. Bir görev her bir olayı işlediğinde bir semaforu 'alacak' (take) ve bu da her 'alma' işleminde semaforun sayım değerinin azalmasına (decremented) neden olacaktır. Sayım değeri, gerçekleşen olayların sayısı ile işlenen olayların sayısı arasındaki farktır. Bu mekanizma Şekil 7.8'de gösterilmektedir.
   Olayları saymak için kullanılan sayıcı semaforlar, sıfır başlangıç sayım değeri ile oluşturulur.
   (Not: Olayları saymak için doğrudan göreve bildirim - direct to task notification - kullanmak, sayıcı bir semafor kullanmaktan daha verimlidir. Doğrudan göreve bildirimler Bölüm 9'a kadar ele alınmamıştır.)
2. **Kaynak yönetimi (Resource management):** Bu senaryoda sayım (count) değeri mevcut kaynakların (resources) sayısını gösterir. Bir kaynağın kontrolünü elde etmek için, görevin önce bir semafor elde etmesi gerekir; bu da semaforun sayım değerini azaltır. Sayım değeri sıfıra ulaştığında boş kaynak kalmamış demektir. Bir görev kaynakla işini bitirdiğinde semaforu geri 'verir', bu da semaforun sayım değerini artırır.
   Kaynakları yönetmek için kullanılan sayıcı semaforlar, ilk sayım değerleri (initial count value) mevcut kaynakların sayısına eşit olacak şekilde oluşturulur. Bölüm 8, kaynakları yönetmek için semaforların kullanılmasını kapsamaktadır.

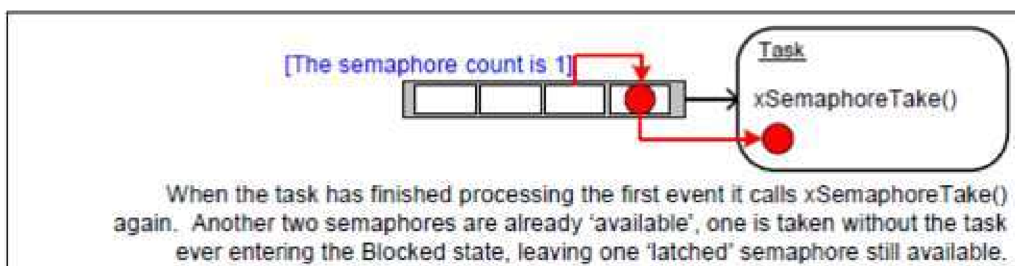
[The semaphore count is 0]
Task
xSemaphoreTake()
The task is blocked waiting for a semaphore


Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....


Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...


[The semaphore count is 0]
Task
vProcessEvent()
...that now successfully 'takes' the semaphore, so it is unavailable once more. The task now starts to process the event.


Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 2]
Task
vProcessEvent()
Another two interrupts occur while the task is still processing the first event. Both ISRs 'give' the semaphore, effectively latching both events, so neither event is lost.
The task is still processing the first event.


[The semaphore count is 1]
Task
xSemaphoreTake()
When the task has finished processing the first event it calls xSemaphoreTake() again. Another two semaphores are already 'available', one is taken without the task ever entering the Blocked state, leaving one 'latched' semaphore still available.

Şekil 7.8 Olayları 'saymak' (count) için bir sayıcı semaforun kullanılması

7.5.1 xSemaphoreCreateCounting() API İşlevi

FreeRTOS ayrıca derleme zamanında (compile time) sayıcı bir semafor oluşturmak için gereken belleği statik olarak tahsis eden `xSemaphoreCreateCountingStatic()` işlevini de içerir. Çeşitli FreeRTOS semafor türlerinin hepsine yönelik tanıtıcılar (handles), `SemaphoreHandle_t` türündeki bir değişkende saklanır.

Bir semafor kullanılmadan önce açıkça (explicitly) oluşturulmalıdır. Sayıcı bir semafor oluşturmak için `xSemaphoreCreateCounting()` API işlevini kullanın.

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,

                                            UBaseType_t uxInitialCount );
```

Liste 7.11 `xSemaphoreCreateCounting()` API işlev prototipi

xSemaphoreCreateCounting() parametreleri ve dönüş değeri

- uxMaxCount

  Semaforun sayacağı maksimum değer. Kuyruk benzetmesiyle, uxMaxCount değeri etkili bir şekilde kuyruğun uzunluğudur.

  Semafor olayları saymak veya tutuklamak (latch) için kullanılacaksa, uxMaxCount tutuklanabilecek maksimum olay sayısıdır.

  Semafor bir kaynak koleksiyonuna erişimi yönetmek için kullanılacaksa, uxMaxCount mevcut toplam kaynak sayısına ayarlanmalıdır.

- uxInitialCount

  Semaforun oluşturulduktan sonraki başlangıç sayım değeri.

  Semafor olayları saymak veya tutuklamak için kullanılacaksa, uxInitialCount sıfır olarak ayarlanmalıdır (çünkü semafor oluşturulduğunda henüz hiçbir olayın gerçekleşmediğini varsayarız).

  Semafor bir kaynak koleksiyonuna erişimi yönetmek için kullanılacaksa, uxInitialCount uxMaxCount'a eşit olarak ayarlanmalıdır (çünkü semafor oluşturulduğunda tüm kaynakların mevcut olduğunu varsayarız).

- Dönüş değeri

  NULL döndürülürse, semafor veri yapılarını ayırmak için yeterli yığın bellek (heap memory) bulunmadığından semafor oluşturulamaz. Bölüm 3, yığın bellek yönetimi hakkında daha fazla bilgi sağlar.

  NULL olmayan bir değer döndürülürse, semaforun başarıyla oluşturulduğunu gösterir. Döndürülen değer, oluşturulan semaforun tanıtıcısı (handle) olarak saklanmalıdır.

Örnek 7.2 Bir görevi bir kesmeyle senkronize etmek için sayıcı semafor kullanma

Örnek 7.2, ikili semafor yerine bir sayıcı semafor kullanarak Örnek 7.1'deki uygulamayı geliştirir. `main()` işlevi `xSemaphoreCreateBinary()` çağrısı yerine `xSemaphoreCreateCounting()` çağrısını içerecek şekilde değiştirilmiştir. Yeni API çağrısı Liste 7.12'de gösterilmektedir.

```
/* Bir semafor kullanılmadan önce açıkça oluşturulmalıdır. Bu örnekte bir
   sayıcı semafor (counting semaphore) oluşturulmaktadır. Semafor, maksimum
   sayım (count) değeri 10 ve başlangıç sayım değeri 0 olacak şekilde
   oluşturulmuştur. */
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

Liste 7.12 Örnek 7.2'de sayıcı semaforu oluşturmak için kullanılan `xSemaphoreCreateCounting()` çağrısı

Yüksek frekansta oluşan birden fazla olayı simüle etmek için, kesme hizmet rutini semaforu kesme başına birden fazla kez 'verecek' şekilde değiştirilmiştir. Her olay, semaforun sayım değerinde tutuklanır (latch edilir). Değiştirilen kesme hizmet rutini Liste 7.13'te gösterilmektedir.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    /* xHigherPriorityTaskWoken parametresi pdFALSE olarak baslatilmalidir
       cunku bir baglam degisikligi gerekiyorsa kesme guvenli API islevi
       icinde pdTRUE olarak ayarlanacaktir. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Semaforu birden fazla kez 'ver'. Ilki ertelenmis kesme isleme
       gorevinin engelini kaldirir, sonraki 'verme'ler kesmelere izin
       veren gorevlerin sirasina gore islenmek uzere ertelenmesini
       saglayarak olaylarin kaybolmadan islenmesini gosterir. Bu,
       islemci tarafindan birden fazla kesme alinmasini simule eder,
       ancak bu durumda olaylar tek bir kesme olusumunda simule edilir. */
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    /* xHigherPriorityTaskWoken degerini portYIELD_FROM_ISR()'a gecir.
       xSemaphoreGiveFromISR() icinde xHigherPriorityTaskWoken pdTRUE olarak
       ayarlandiysa portYIELD_FROM_ISR() cagrisi bir baglam degisikligi
       talep eder. xHigherPriorityTaskWoken hala pdFALSE ise
       portYIELD_FROM_ISR() cagrisinin hicbir etkisi olmaz. Cogu FreeRTOS
       portunun aksine, Windows portu ISR'nin bir deger dondurmesini
       gerektirir – return ifadesi portYIELD_FROM_ISR()'in Windows
       versiyonunun icindedir. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Liste 7.13 *Örnek 7.2 tarafından kullanılan kesme hizmet rutininin uygulaması*

Diğer tüm işlevler Örnek 7.1'de kullanılanlardan değiştirilmemiştir.
Örnek 7.2 yürütüldüğünde üretilen çıktı Şekil 7.9'da gösterilmektedir. Görülebileceği gibi, kesme işlemenin ertelendiği görev, bir kesme oluşturulduğunda her seferinde üç (simüle edilmiş) olayı işlemektedir. Olaylar, semaforun sayım değerinde tutuklanarak görevin bunları sırayla işlemesine olanak tanır.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

Şekil 7.9 *Örnek 7.2 yürütüldüğünde üretilen çıktı*

7.6 İşi RTOS Daemon Görevine Erteleme

Şimdiye kadar sunulan ertelenmiş kesme işleme örnekleri, uygulama yazarının ertelenmiş işleme tekniğini kullanan her kesme için bir görev oluşturmasını gerektirmiştir. Kesme işlemeyi RTOS daemon görevine ertelemek için xTimerPendFunctionCallFromISR()[^19] API işlevini kullanmak da mümkündür ve bu, her kesme için ayrı bir görev oluşturma gerekliliğini ortadan kaldırır. Kesme işlemeyi daemon görevine ertelemeye 'merkezileştirilmiş ertelenmiş kesme işleme' denir.

[^19]: Daemon görevi başlangıçta yalnızca yazılım zamanlayıcı geri çağırma işlevlerini yürütmek için kullanıldığından, zamanlayıcı hizmet görevi olarak adlandırılıyordu. Bu nedenle, xTimerPendFunctionCall() timers.c'de uygulanmıştır ve adlandırma kuralına uygun olarak işlev adı 'Timer' ön ekini almaktadır.

Bölüm 6, yazılım zamanlayıcıyla ilgili FreeRTOS API işlevlerinin zamanlayıcı komut kuyruğu üzerinden daemon görevine nasıl komut gönderdiğini açıklamıştır.

xTimerPendFunctionCall() ve xTimerPendFunctionCallFromISR() API işlevleri, aynı zamanlayıcı komut kuyruğunu kullanarak daemon görevine bir 'işlev yürüt' komutu gönderir. Daemon görevine gönderilen işlev daha sonra daemon görevinin bağlamında yürütülür.
Merkezileştirilmiş ertelenmiş kesme işlemenin avantajları şunlardır:

• Daha düşük kaynak kullanımı — Her ertelenmiş kesme için ayrı bir görev oluşturma gerekliliğini ortadan kaldırır.

• Basitleştirilmiş kullanıcı modeli — Ertelenmiş kesme işleme işlevi standart bir C işlevidir.

Merkezileştirilmiş ertelenmiş kesme işlemenin dezavantajları şunlardır:

• Daha az esneklik — Her ertelenmiş kesme işleme görevinin önceliğini ayrı ayrı ayarlamak mümkün değildir. Her ertelenmiş kesme işleme işlevi daemon görevinin önceliğinde yürütülür. Bölüm 6'da açıklandığı gibi, daemon görevinin önceliği FreeRTOSConfig.h'daki

configTIMER_TASK_PRIORITY derleme zamanı yapılandırma sabiti tarafından ayarlanır.
• Daha az determinizm — xTimerPendFunctionCallFromISR() zamanlayıcı komut kuyruğunun arkasına bir komut gönderir. Zamanlayıcı komut kuyruğunda zaten bulunan komutlar, xTimerPendFunctionCallFromISR() tarafından kuyruğa gönderilen 'işlev yürüt' komutundan önce daemon görevi tarafından işlenecektir.

Farklı kesmelerin farklı zamanlama kısıtlamaları olduğundan, aynı uygulama içinde her iki ertelenmiş kesme işleme yönteminin de kullanılması yaygındır.

7.6.1 xTimerPendFunctionCallFromISR() API İşlevi

xTimerPendFunctionCallFromISR(), xTimerPendFunctionCall()'ın kesme güvenli versiyonudur. Her iki API işlevi de uygulama yazarı tarafından sağlanan bir işlevin daemon görevinin bağlamında yürütülmesine olanak tanır. Yürütülecek işlev ve işlevin giriş parametrelerinin değeri, zamanlayıcı komut kuyruğu üzerinden daemon görevine gönderilir. İşlevin gerçekten ne zaman yürütüleceği, uygulamadaki diğer görevlere göre daemon görevinin önceliğine bağlıdır.

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t
                                          xFunctionToPend,
                                          void *pvParameter1,
                                          uint32_t ulParameter2,
                                          BaseType_t *pxHigherPriorityTaskWoken
);
```

Liste 7.14 *xTimerPendFunctionCallFromISR() API işlev prototipi*

xTimerPendFunctionCallFromISR()'ın xFunctionToPend parametresinde geçirilen bir işlevin uyması gereken prototip Liste 7.15'te gösterilmektedir.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

Liste 7.15 *xTimerPendFunctionCallFromISR()'ın xFunctionToPend parametresinde geçirilen bir işlevin uyması gereken prototip*

xTimerPendFunctionCallFromISR() parametreleri ve dönüş değeri

• xFunctionToPend
Daemon görevinde yürütülecek işleve bir işaretçi (gerçekte sadece işlev adı). İşlevin prototipi Liste 7.15'te gösterilenle aynı olmalıdır.

• pvParameter1
Daemon görevi tarafından yürütülen işleve, işlevin pvParameter1 parametresi olarak geçirilecek değer. Parametrenin herhangi bir veri türünü geçirmeye olanak tanıyan bir void * türü vardır. Örneğin, tamsayı türleri doğrudan void *'a dönüştürülebilir veya alternatif olarak void * bir yapıya işaret etmek için kullanılabilir.

• ulParameter2
Daemon görevi tarafından yürütülen işleve, işlevin ulParameter2 parametresi olarak geçirilecek değer.

• pxHigherPriorityTaskWoken
xTimerPendFunctionCallFromISR() zamanlayıcı komut kuyruğuna yazar. RTOS daemon görevi zamanlayıcı komut kuyruğunda verinin kullanılabilir olmasını bekleyerek Engellenmiş durumdaysa, zamanlayıcı komut kuyruğuna yazmak daemon görevinin Engellenmiş durumdan çıkmasına neden olacaktır. Daemon görevinin önceliği, o anda yürütülen görevinkinden (kesintiye uğrayan görev) yüksekse, xTimerPendFunctionCallFromISR() dahili olarak *pxHigherPriorityTaskWoken'ı pdTRUE olarak ayarlayacaktır.

• Dönüş değeri — İki olası dönüş değeri vardır:
  ○ pdPASS — 'İşlev yürüt' komutu zamanlayıcı komut kuyruğuna yazıldığında döndürülür.
  ○ pdFAIL — Zamanlayıcı komut kuyruğu zaten dolu olduğu için 'işlev yürüt' komutu zamanlayıcı komut kuyruğuna yazılamadığında döndürülür. Bölüm 6, zamanlayıcı komut kuyruğunun uzunluğunun nasıl ayarlanacağını açıklamaktadır.

Örnek 7.3 *Merkezileştirilmiş ertelenmiş kesme işleme*

Örnek 7.3, Örnek 7.1'e benzer işlevsellik sağlar, ancak bir semafor kullanmadan ve kesme tarafından zorunlu kılınan işlemi gerçekleştirmek için özel bir görev oluşturmadan. Bunun yerine, işlem RTOS daemon görevi tarafından gerçekleştirilir.

Örnek 7.3 tarafından kullanılan kesme hizmet rutini Liste 7.16'da gösterilmektedir. xTimerPendFunctionCallFromISR()'ı kullanarak vDeferredHandlingFunction() adlı bir işleve bir işaretçiyi daemon görevine geçirir. Ertelenmiş kesme işleme vDeferredHandlingFunction() işlevi tarafından gerçekleştirilir.

Kesme hizmet rutini her yürütüldüğünde ulParameterValue adlı bir değişkeni artırır. ulParameterValue, xTimerPendFunctionCallFromISR() çağrısında ulParameter2 değeri olarak kullanılır, bu nedenle vDeferredHandlingFunction() daemon görevi tarafından yürütüldüğünde ulParameter2 değeri olarak da kullanılacaktır. İşlevin diğer parametresi olan pvParameter1 bu örnekte kullanılmamaktadır.

```
static uint32_t ulExampleInterruptHandler( void )
{
    static uint32_t ulParameterValue = 0;
    BaseType_t xHigherPriorityTaskWoken;

    /* xHigherPriorityTaskWoken parametresi pdFALSE olarak baslatilmalidir
       cunku bir baglam degisikligi gerekiyorsa kesme guvenli API islevi
       icinde pdTRUE olarak ayarlanacaktir. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Kesmenin ertelenmis isleme islevine bir isaretci daemon gorevine
       gonderin. Ertelenmis isleme islevinin pvParameter1 parametresi
       kullanilmiyor, bu yuzden sadece NULL olarak ayarlandi. Ertelenmis
       isleme islevinin ulParameter2 parametresi, bu kesme isleyicisi
       her yurutuldugunde bir artan bir sayi gecirmek icin kullanilir. */
    xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Yurutulecek
islev */
                                   NULL, /* Kullanilmiyor */
                                   ulParameterValue, /* Artan deger. */
                                   &xHigherPriorityTaskWoken );
    ulParameterValue++;

    /* xHigherPriorityTaskWoken degerini portYIELD_FROM_ISR()'a gecir.
       xTimerPendFunctionCallFromISR() icinde xHigherPriorityTaskWoken
       pdTRUE olarak ayarlandiysa portYIELD_FROM_ISR() cagrisi bir
       baglam degisikligi talep eder. xHigherPriorityTaskWoken hala
       pdFALSE ise portYIELD_FROM_ISR() cagrisinin hicbir etkisi olmaz.
       Cogu FreeRTOS portunun aksine, Windows portu ISR'nin bir deger
       dondurmesini gerektirir – return ifadesi portYIELD_FROM_ISR()'in
       Windows versiyonunun icindedir. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Liste 7.16 *Örnek 7.3'te kullanılan yazılım kesmesi işleyicisi*

vDeferredHandlingFunction()'ın uygulaması Liste 7.17'de gösterilmektedir. Sabit bir dize ve ulParameter2 parametresinin değerini yazdırır.

vDeferredHandlingFunction(), bu örnekte parametrelerinden yalnızca biri kullanılsa da Liste 7.15'te gösterilen prototipe sahip olmalıdır.

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t
ulParameter2 )
{
    /* Olayi isle – bu durumda sadece bir mesaj ve ulParameter2'nin
       degerini yazdir. pvParameter1 bu ornekte kullanilmiyor. */
    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2
);
}
```

Liste 7.17 *Örnek 7.3'te kesme tarafından zorunlu kılınan işlemeyi gerçekleştiren işlev*

Örnek 7.3 tarafından kullanılan main() işlevi Liste 7.18'de gösterilmektedir. Ertelenmiş kesme işlemeyi gerçekleştirmek için ne bir semafor ne de bir görev oluşturmadığından, Örnek 7.1 tarafından kullanılan main() işlevinden daha basittir.

vPeriodicTask() periyodik olarak yazılım kesmesi oluşturan görevdir. Daemon görevinin Engellenmiş durumdan çıkar çıkmaz oncelemesini sağlamak için daemon görevinin önceliğinin altında bir öncelikle oluşturulur.

```
int main( void )
{
    /* Yazilim kesmesini ureten gorev, daemon gorevinin onceligi
       altinda bir oncelikle olusturulur. Daemon gorevinin onceligi
       FreeRTOSConfig.h'daki configTIMER_TASK_PRIORITY derleme zamani
       yapilandirma sabiti tarafindan ayarlanir. */
    const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Periyodik olarak yazilim kesmesi uretecek gorevi olusturun. */
    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority,
                 NULL );

    /* Yazilim kesmesi icin isleyiciyi kurun. Bunu yapmak icin gerekli
       sozdizimi, kullanilan FreeRTOS portuna baglidir. Burada gosterilen
       sozdizimi yalnizca bu kesmelerin sadece simule edildigi FreeRTOS
       Windows portu ile kullanilabilir. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler
);

    /* Cizgeleyiciyi baslatin, boylece olusturulan gorev yurutulmeye baslar. */
    vTaskStartScheduler();

    /* Her zamanki gibi, asagidaki satira asla ulasilmamalidir. */
    for( ;; );
}
```

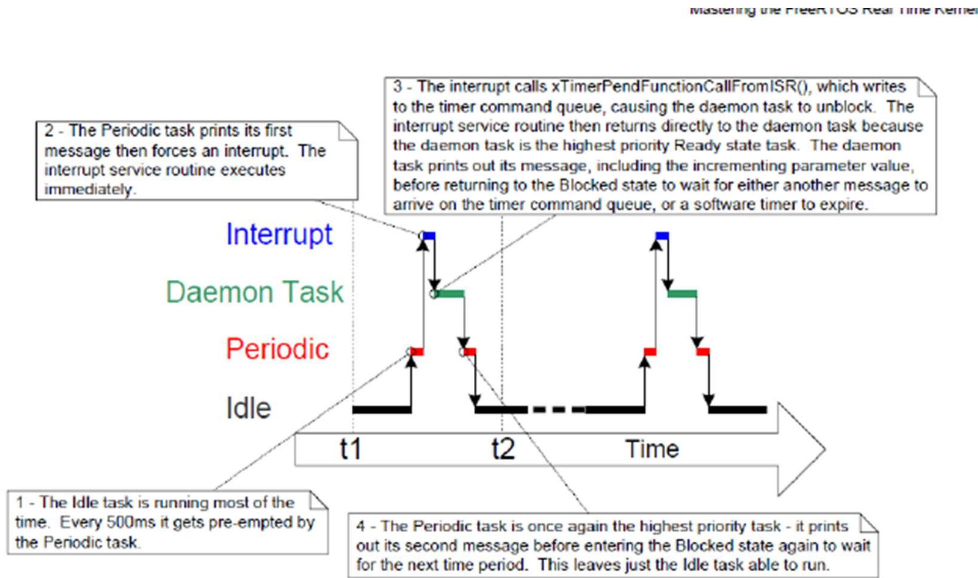
Liste 7.18 *Örnek 7.3 için main() uygulaması*

Örnek 7.3, Şekil 7.10'da gösterilen çıktıyı üretir. Daemon görevinin önceliği, yazılım kesmesini oluşturan görevin önceliğinden yüksektir, bu nedenle vDeferredHandlingFunction() kesme oluşturulur oluşturulmaz daemon görevi tarafından yürütülür.

Bunun sonucunda vDeferredHandlingFunction() tarafından çıktılanan mesaj, periyodik görev tarafından çıktılanan iki mesajın arasında görünür, tıpkı özel bir ertelenmiş kesme işleme görevi bir semaforu engelini kaldırmak için kullanıldığında olduğu gibi. Daha fazla açıklama Şekil 7.11'de verilmektedir.

```
C:\Windows\system32\cmd.exe - rtosdemo

C:\temp>rtosdemo
Periodic task - About to generate an interrupt.
Handler function - Processing event  0
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  1
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  2
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event  3
Periodic task - Interrupt generated.
```

Şekil 7.10 *Örnek 7.3 yürütüldüğünde üretilen çıktı*



Şekil 7.11 *Örnek 7.3 yürütüldüğünde yürütme sırası*

7.7 Bir Kesme Hizmet Rutini İçinde Kuyrukların Kullanılması

İkili ve sayma semaforları olayları iletmek için kullanılır. Kuyruklar hem olayları iletmek hem de veri aktarmak için kullanılır.

xQueueSendToFrontFromISR(), bir kesme hizmet rutininde güvenle kullanılabilen xQueueSendToFront() versiyonudur, xQueueSendToBackFromISR(), bir kesme hizmet rutininde güvenle kullanılabilen xQueueSendToBack() versiyonudur ve xQueueReceiveFromISR(), bir kesme hizmet rutininde güvenle kullanılabilen xQueueReceive() versiyonudur.

7.7.1 xQueueSendToFrontFromISR() ve xQueueSendToBackFromISR() API İşlevleri

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue,
                                     const void *pvItemToQueue
                                     BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 7.19 *xQueueSendToFrontFromISR() API işlev prototipi*

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue,
                                    const void *pvItemToQueue
                                    BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 7.20 *xQueueSendToBackFromISR() API işlev prototipi*

xQueueSendFromISR() ve xQueueSendToBackFromISR() işlevsel olarak eşdeğerdir.
xQueueSendToFrontFromISR() ve xQueueSendToBackFromISR() parametreleri ve dönüş değerleri

- xQueue
  Verinin gönderildiği (yazıldığı) kuyruğun tanıtıcısı (handle). Kuyruk tanıtıcısı, kuyruğu oluşturmak için kullanılan xQueueCreate() çağrısından döndürülmüş olacaktır.

- pvItemToQueue
  Kuyruğa yerleştirilecek öğeye bir işaretçi. Kuyruğun tutacağı her öğenin boyutu kuyruk oluşturulduğunda tanımlandı, bu nedenle bu kadar bayt pvItemToQueue'dan kuyruk depolama alanına kopyalanacaktır.

- pxHigherPriorityTaskWoken
  Tek bir kuyrukta bir veya daha fazla görevin, verinin kullanılabilir olmasını bekleyerek engellenmiş olması mümkündür. xQueueSendToFrontFromISR() veya xQueueSendToBackFromISR() çağrısı veriyi kullanılabilir hale getirebilir ve böylece bir görevin Engellenmiş durumdan çıkmasına neden olabilir. API işlevi dahili olarak *pxHigherPriorityTaskWoken'ı pdTRUE olarak ayarlayacaktır.

- Dönüş değeri — İki olası dönüş değeri vardır:
  ○ pdPASS — Yalnızca veri kuyruğa başarıyla gönderildiğinde döndürülür.
  ○ errQUEUE_FULL — Kuyruk zaten dolu olduğu için veri kuyruğa gönderilemediğinde döndürülür.

7.7.2 Bir ISR'den Kuyruk Kullanırken Dikkat Edilecekler

Kuyruklar, bir kesmeden bir göreve veri aktarmanın kolay ve kullanışlı bir yolunu sağlar, ancak veri yüksek frekansta geliyorsa bir kuyruk kullanmak verimli değildir.

FreeRTOS indirmesindeki demo uygulamaların çoğu, UART'ın alım ISR'sinden karakterleri aktarmak için kuyruk kullanan basit bir UART sürücüsü içerir. Bu demolarda, bir ISR'den kuyruk kullanımını göstermek ve sistemi kasıtlı olarak yüklemek için iki nedenden dolayı kuyruk kullanılmaktadır. Bir kuyruğu bu şekilde kullanan ISR'ler kesinlikle verimli bir tasarımı temsil etmez ve veri yavaş gelmiyorsa, üretim kodunun bu tekniği kopyalamaması önerilir. Üretim kodu için uygun daha verimli teknikler şunlardır:

• Doğrudan Bellek Erişimi (DMA) donanımı kullanarak karakterleri almak ve tamponlamak. Bu yöntemin neredeyse hiç yazılım yükü yoktur. Bir doğrudan görev bildirimi[^20], yalnızca iletimde bir kesinti algılandıktan sonra tamponu işleyecek görevi engelini kaldırmak için kullanılabilir.

• Alınan her karakteri iş parçacığı güvenli (thread safe) bir RAM tamponuna kopyalamak[^21]. Yine, tam bir mesaj alındıktan veya iletimde bir kesinti algılandıktan sonra tamponu işleyecek görevi engelini kaldırmak için bir doğrudan görev bildirimi kullanılabilir.
• Alınan karakterleri doğrudan ISR içinde işlemek, ardından işlemenin sonucunu (ham veri yerine) bir göreve göndermek için kuyruk kullanmak. Bu daha önce Şekil 5.4 ile gösterilmiştir.

Örnek 7.4 *Bir kesme içinden bir kuyrukta gönderme ve alma*
Bu örnek, aynı kesme içinde kullanılan xQueueSendToBackFromISR() ve xQueueReceiveFromISR()'ı gösterir. Daha önce olduğu gibi, kolaylık olması için kesme yazılım tarafından oluşturulur.

Her 200 milisaniyede bir kuyruğa beş sayı gönderen periyodik bir görev oluşturulur. Beş değerin tamamı gönderildikten sonra bir yazılım kesmesi oluşturur. Görev uygulaması Liste 7.21'de gösterilmektedir.

```
static void vIntegerGenerator( void *pvParameters )
{
    TickType_t xLastExecutionTime;
    uint32_t ulValueToSend = 0;
    int i;

    /* vTaskDelayUntil() cagrisinda kullanilan degiskeni baslatin. */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* Bu periyodik bir gorevdir. Tekrar calisma zamani gelene
           kadar engelle. Gorev her 200ms'de bir yurutulecektir. */
        vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );

        /* Kuyruga bes sayi gonderin, her deger oncekinden bir fazla.
           Sayilar kuyruktan kesme hizmet rutini tarafindan okunur.
           Kesme hizmet rutini her zaman kuyrugu bosaltir, bu nedenle
```

```
            bu gorev engelleme suresi belirtmeye gerek kalmadan bes
            degerin tamamini yazabilmesi garantidir. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }

        /* Kesmeyi uretin, boylece kesme hizmet rutini degerleri
           kuyruktan okuyabilir. Yazilim kesmesi uretmek icin
           kullanilan sozdizimi, kullanilan FreeRTOS portuna baglidir.
           Asagida kullanilan sozdizimi yalnizca bu kesmelerin sadece
           simule edildigi FreeRTOS Windows portu ile kullanilabilir. */
        vPrintString( "Generator task - About to generate an interrupt.\r\n" );
        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Liste 7.21 *Örnek 7.4'te kuyruğa yazan görevin uygulaması*

Kesme hizmet rutini, periyodik görev tarafından kuyruğa yazılan tüm değerler okunana ve kuyruk boşalana kadar tekrar tekrar xQueueReceiveFromISR()'ı çağırır. Alınan her değerin son iki biti, bir dize dizisine indeks olarak kullanılır. Karşılık gelen indeks konumundaki dizeye bir işaretçi, xQueueSendFromISR() çağrısı kullanılarak farklı bir kuyruğa gönderilir. Kesme hizmet rutininin uygulaması Liste 7.22'de gösterilmektedir.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;
    uint32_t ulReceivedNumber;

    /* Dizeler, kesme hizmet rutininin yigininda (stack) ayrilmadigindan
       emin olmak icin static const olarak tanimlanir ve dolayisiyla
       kesme hizmet rutini yurutulmediginde bile var olur. */
    static const char *pcStrings[] =
    {
        "String 0\r\n",
        "String 1\r\n",
        "String 2\r\n",
        "String 3\r\n"
    };

    /* Her zamanki gibi, xHigherPriorityTaskWoken pdFALSE olarak
       baslatilir, boylece kesme guvenli bir API islevi icinde
       pdTRUE'ya ayarlandiginin tespit edilebilmesi saglanir.
       Kesme guvenli bir API islevi yalnizca xHigherPriorityTaskWoken'i
       pdTRUE olarak ayarlayabildiginden, hem xQueueReceiveFromISR()
       hem de xQueueSendToBackFromISR() cagrisinda ayni
       xHigherPriorityTaskWoken degiskenini kullanmak guvenlidir. */
    xHigherPriorityTaskWoken = pdFALSE;
```

```
    /* Kuyruk bos olana kadar kuyruktan oku. */
    while( xQueueReceiveFromISR( xIntegerQueue,
                                 &ulReceivedNumber,
                                 &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY
)
    {
        /* Alinan degeri son iki bite kir (0'dan 3'e kadar degerler
           dahil), ardından kırpılmış degeri pcStrings[] dizisine
           indeks olarak kullanarak diger kuyruga gonderilecek bir
           dize (char *) secin. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                 &pcStrings[ ulReceivedNumber ],
                                 &xHigherPriorityTaskWoken );
    }

    /* xIntegerQueue'dan alma bir gorevin Engellenmiş durumdan
       cikmasina neden olduysa ve Engellenmiş durumdan cikan gorevin
       onceligi Calisiyor durumundaki gorevinkinden yuksekse,
       xHigherPriorityTaskWoken xQueueReceiveFromISR() icinde
       pdTRUE olarak ayarlanmis olacaktir.

       xStringQueue'a gonderme bir gorevin Engellenmiş durumdan
       cikmasina neden olduysa ve Engellenmiş durumdan cikan gorevin
       onceligi Calisiyor durumundaki gorevinkinden yuksekse,
       xHigherPriorityTaskWoken xQueueSendToBackFromISR() icinde
       pdTRUE olarak ayarlanmis olacaktir.

       xHigherPriorityTaskWoken portYIELD_FROM_ISR()'a parametre
       olarak kullanilir. xHigherPriorityTaskWoken pdTRUE ise
       portYIELD_FROM_ISR() cagrisi bir baglam degisikligi talep
       eder. xHigherPriorityTaskWoken hala pdFALSE ise
       portYIELD_FROM_ISR() cagrisinin hicbir etkisi olmaz.

       Windows portu tarafindan kullanilan portYIELD_FROM_ISR()
       uygulamasi bir return ifadesi icerir, bu nedenle bu islev
       acikca bir deger dondurmez. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

***Liste** 7.22 Örnek 7.4 tarafından kullanılan kesme hizmet rutininin uygulaması*

Kesme hizmet rutininden karakter işaretçilerini alan görev, bir mesaj gelene kadar kuyrukta engellenir ve aldığı her dizeyi yazdırır. Uygulaması Liste 7.23'te gösterilmektedir.

```
static void vStringPrinter( void *pvParameters )
{
    char *pcString;

    for( ;; )
```

```
    {
        /* Verinin gelmesini beklemek icin kuyrukta engelle. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Alinan dizeyi yazdir. */
        vPrintString( pcString );
    }
}
```

***Liste** 7.23 Örnek 7.4'te kesme hizmet rutininden alınan dizeleri yazdıran görev*

Her zamanki gibi, main() çizgeleyiciyi başlatmadan önce gerekli kuyrukları ve görevleri oluşturur. Uygulaması Liste 7.24'te gösterilmektedir.

```
int main( void )
{
    /* Bir kuyruk kullanilmadan once olusturulmalidir. Bu ornek
       tarafindan kullanilan her iki kuyrugu olusturun. Bir kuyruk
       uint32_t turunde degiskenleri tutabilir, digeri char* turunde
       degiskenleri tutabilir. Her iki kuyruk da maksimum 10 oge
       tutabilir. Gercek bir uygulama kuyrukların basariyla
       olusturuldugundan emin olmak icin donus degerlerini kontrol
       etmelidir. */
    xIntegerQueue = xQueueCreate( 10, sizeof( uint32_t ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Kesme hizmet rutinine tamsayilari gecirmek icin kuyruk kullanan
       gorevi olusturun. Gorev oncelik 1 ile olusturulur. */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* Kesme hizmet rutininden kendisine gonderilen dizeleri yazdiran
       gorevi olusturun. Bu gorev daha yuksek 2 onceligi ile
       olusturulur. */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* Yazilim kesmesi icin isleyiciyi kurun. Bunu yapmak icin gerekli
       sozdizimi, kullanilan FreeRTOS portuna baglidir. Burada
       gosterilen sozdizimi yalnizca bu kesmelerin sadece simule
       edildigi FreeRTOS Windows portu ile kullanilabilir. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler
);

    /* Cizgeleyiciyi baslatin, boylece olusturulan gorevler
       yurutulmeye baslar. */
    vTaskStartScheduler();

    /* Her sey yolundaysa main() buraya asla ulasamaz cünkü cizgeleyici
       artik görevleri yurutuyor olacaktir. main() buraya ulasirsa
       bosta gorevi olusturmak icin yeterli yigin bellek (heap memory)
       olmadigini gosterir. Bolum 2, yigin bellek yonetimi hakkinda
       daha fazla bilgi saglar. */
    for( ;; );
```

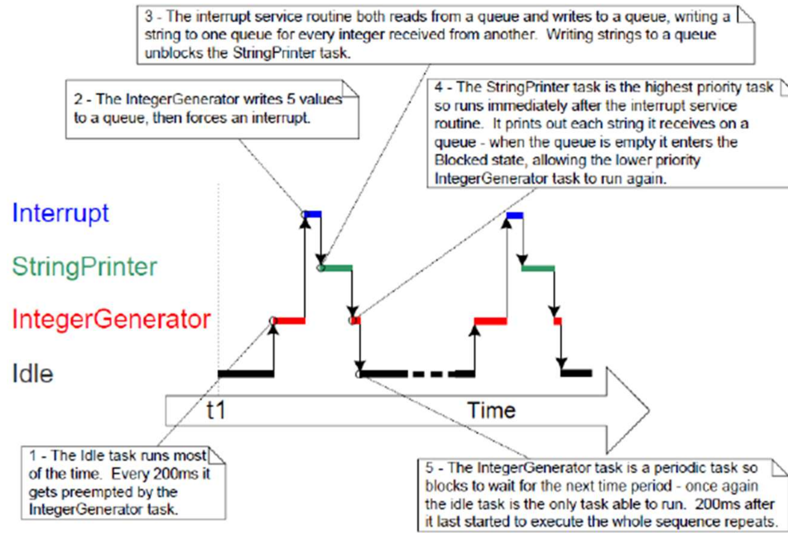
```
}
```

__Liste 7.24__ Örnek 7.4 için main() işlevi

Örnek 7.4 yürütüldüğünde üretilen çıktı Şekil 7.12'de gösterilmektedir. Görülebileceği gibi, kesme beş tamsayının tamamını alır ve yanıt olarak beş dize üretir. Daha fazla açıklama Şekil 7.13'te verilmektedir.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
String 3
String 0
String 1
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 2
String 3
String 0
String 1
String 2
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 3
String 0
String 1
String 2
String 3
Generator task - Interrupt generated.
```

__Şekil 7.12__ Örnek 7.4 yürütüldüğünde üretilen çıktı



__Şekil 7.13__ Örnek 7.4 tarafından üretilen yürütme sırası

7.8 Kesme Yuvalama (Interrupt Nesting)

Görev öncelikleri (task priorities) ile kesme öncelikleri (interrupt priorities) arasında kafa karışıklığı yaşanması yaygındır. Bu bölüm, kesme servis rutinlerinin (ISR'ler) birbirlerine göre yürütülme öncelikleri olan kesme önceliklerini tartışmaktadır. Bir göreve atanan önceliğin bir kesmeye atanan öncelikle hiçbir ilgisi yoktur. Donanım (hardware), bir ISR'nin ne zaman çalışacağına karar verirken, yazılım (software), bir görevin ne zaman çalışacağına karar verir. Bir donanım kesmesine yanıt olarak yürütülen bir ISR bir görevi kesintiye uğratır (interrupt), ancak bir görev bir ISR'yi engelleyemez (pre-empt yapamaz).

Kesme yuvalamasını (interrupt nesting) destekleyen portlar, `FreeRTOSConfig.h`'de aşağıda ayrıntıları verilen sabitlerden birinin veya her ikisinin tanımlanmasını gerektirir. `configMAX_SYSCALL_INTERRUPT_PRIORITY` ve `configMAX_API_CALL_INTERRUPT_PRIORITY` ikisi de aynı özelliği tanımlar. Eski FreeRTOS portları `configMAX_SYSCALL_INTERRUPT_PRIORITY` kullanır, daha yeni FreeRTOS portları ise `configMAX_API_CALL_INTERRUPT_PRIORITY` kullanır.

Kesme yuvalamasını kontrol eden sabitler:

- **`configMAX_SYSCALL_INTERRUPT_PRIORITY`** veya **`configMAX_API_CALL_INTERRUPT_PRIORITY`**: Kesme güvenli (interrupt-safe) FreeRTOS API işlevlerinin çağrılabileceği en yüksek kesme önceliğini ayarlar.
- **`configKERNEL_INTERRUPT_PRIORITY`**: Tick kesmesi (tick interrupt) tarafından kullanılan kesme önceliğini ayarlar ve her zaman mümkün olan en düşük kesme önceliğine ayarlanmalıdır.

Kullanılan FreeRTOS portu ayrıca `configMAX_SYSCALL_INTERRUPT_PRIORITY` sabitini de kullanmıyorsa, kesme güvenli FreeRTOS API işlevlerini kullanan herhangi bir kesmenin ayrıca `configKERNEL_INTERRUPT_PRIORITY` tarafından tanımlanan öncelikte yürütülmesi gerekir.

Her kesme kaynağının bir **sayısal (numeric)** önceliği ve bir **mantıksal (logical)** önceliği vardır:

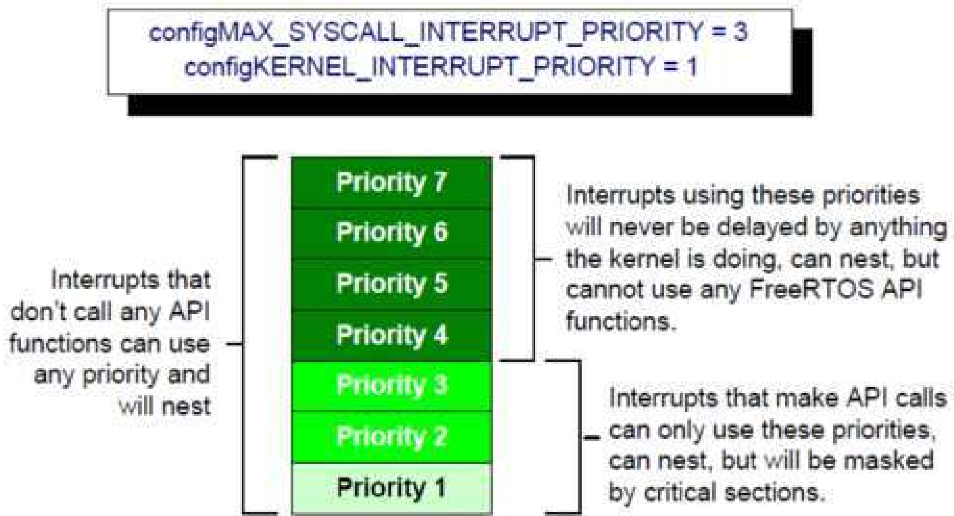
- **Sayısal öncelik (Numeric priority):** Sayısal öncelik, basitçe kesme önceliğine atanan sayıdır. Örneğin, bir kesmeye 7 önceliği atanırsa, sayısal önceliği 7'dir. Benzer şekilde, bir kesmeye 200 önceliği atanırsa, sayısal önceliği 200'dür.
- **Mantıksal öncelik (Logical priority):** Bir kesmenin mantıksal önceliği, o kesmenin diğer kesmelere göre önceliğini (precedence) tanımlar. Farklı öncelikteki iki kesme aynı anda meydana gelirse, işlemci daha düşük mantıksal önceliğe sahip kesmenin ISR'sini yürütmeden önce daha yüksek mantıksal önceliğe sahip kesmenin ISR'sini yürütecektir.

  Bir kesme, daha düşük mantıksal önceliğe sahip herhangi bir kesmeyi kesintiye uğratabilir (onunla yuvalanabilir - nest with), ancak bir kesme, eşit veya daha yüksek mantıksal önceliğe sahip hiçbir kesmeyi kesintiye uğratamaz (onunla yuvalanamaz).

Bir kesmenin sayısal önceliği ile mantıksal önceliği arasındaki ilişki işlemci mimarisine bağlıdır; bazı işlemcilerde, bir kesmeye atanan sayısal öncelik ne kadar yüksekse o kesmenin mantıksal önceliği de o kadar yüksek olacaktır, diğer işlemci mimarilerinde ise bir kesmeye atanan sayısal öncelik ne kadar yüksekse o kesmenin mantıksal önceliği o kadar düşük olacaktır.

Tam (full) bir kesme yuvalama modeli, `configMAX_SYSCALL_INTERRUPT_PRIORITY` sabitinin `configKERNEL_INTERRUPT_PRIORITY`'den daha yüksek mantıksal kesme önceliğine ayarlanmasıyla oluşturulur. Bu, aşağıdaki durumların yaşandığı bir senaryoyu gösteren Şekil 7.14'te gösterilmektedir:

- İşlemcinin yedi benzersiz (unique) kesme önceliği vardır.
- Sayısal önceliği 7 olan kesmeler, sayısal önceliği 1 olan kesmelerden daha yüksek mantıksal önceliğe sahiptir.
- `configKERNEL_INTERRUPT_PRIORITY` bir olarak ayarlanmıştır.
- `configMAX_SYSCALL_INTERRUPT_PRIORITY` üç olarak ayarlanmıştır.



Şekil 7.14 Kesme yuvalama davranışını etkileyen sabitler

Şekil 7.14 referans alındığında:

- 1 ile 3 (dahil) arasındaki öncelikleri kullanan kesmelerin çekirdek (kernel) veya uygulama kritik bir bölümün (critical section) içindeyken yürütülmesi engellenir. Bu önceliklerde çalışan ISR'ler, kesme güvenli FreeRTOS API işlevlerini kullanabilir. Kritik bölümler Bölüm 7'de (sonraki bölümde) açıklanmaktadır.
- 4 veya daha yüksek öncelik kullanan kesmeler kritik bölümlerden etkilenmez, bu nedenle donanımın kendi sınırlamaları dahilinde çizgeleyicinin yapacağı hiçbir şey bu kesmelerin anında yürütülmesini engellemeyecektir. Bu önceliklerde yürütülen ISR'ler herhangi bir FreeRTOS API işlevi kullanamazlar.

- Tipik olarak, çok katı çizgeleme doğruluğu (motor kontrolü gibi) gerektiren işlevsellik, çizgeleyicinin kesme tepki süresine bir seğirme (jitter) eklememesini sağlamak için `configMAX_SYSCALL_INTERRUPT_PRIORITY` sabitinin üzerinde bir öncelik kullanacaktır.

7.8.1 ARM Cortex-M ve ARM GIC Kullanıcılarına Bir Not

(Not: Bu bölüm Cortex-M0 ve Cortex-M0+ çekirdekleri için yalnızca kısmen geçerlidir.)

Cortex-M işlemcilerindeki kesme yapılandırması kafa karıştırıcıdır ve hataya açıktır. Geliştirmenize yardımcı olmak için, FreeRTOS Cortex-M portları kesme yapılandırmasını otomatik olarak kontrol eder, ancak yalnızca `configASSERT()` tanımlanmışsa. `configASSERT()` bölüm 11.2'de açıklanmıştır.

ARM Cortex çekirdekleri ve ARM Genel Kesme Denetleyicileri (Generic Interrupt Controllers - GIC'ler), mantıksal olarak yüksek öncelikli kesmeleri temsil etmek için sayısal olarak düşük öncelik numaraları kullanır. Bu durum sezgilere aykırı (counter-intuitive) görünebilir ve unutulması kolaydır. Bir kesmeye mantıksal olarak düşük bir öncelik atamak istiyorsanız, o zaman sayısal olarak yüksek bir değer atanmalıdır. Bir kesmeye mantıksal olarak yüksek bir öncelik atamak istiyorsanız, o zaman sayısal olarak düşük bir değer atanmalıdır.

Cortex-M kesme denetleyicisi, her bir kesme önceliğini belirlemek için en fazla sekiz bitin kullanılmasına olanak tanır ve böylece 255 olası en düşük öncelik haline gelir. Sıfır en yüksek önceliktir. Ancak, Cortex-M mikrodenetleyicileri normalde olası sekiz bitin yalnızca bir alt kümesini (subset) uygular (implement). Aslında uygulanan bit sayısı mikrodenetleyici ailesine bağlıdır.

Olası sekiz bitin yalnızca bir alt kümesi uygulandığında, baytın yalnızca en anlamlı bitleri (most significant bits) kullanılabilir, en anlamsız bitler (least significant bits) ise uygulanmamış olarak kalır. Uygulanmayan bitler herhangi bir değeri alabilir, ancak bunların 1 olarak ayarlanması normaldir. Bu durum, dört öncelik biti uygulayan bir Cortex-M mikrodenetleyici tarafından 101 ikili (binary) önceliğinin nasıl depolandığını gösteren Şekil 7.15'te gösterilmiştir.

Şekil 7.15 İkili 101 önceliğinin, dört öncelik biti uygulayan bir Cortex-M mikrodenetleyici tarafından nasıl saklandığı

Şekil 7.15'te 101 ikili değeri en anlamlı dört bite kaydırılmıştır (shifted) çünkü en anlamsız dört bit uygulanmamıştır. Uygulanmayan bitler 1 olarak ayarlanmıştır.

Bazı kütüphane işlevleri, öncelik değerlerinin uygulanan (en anlamlı) bitlere kaydırıldıktan sonra belirtilmesini bekler. Böyle bir işlev kullanılırken, Şekil 7.15'te gösterilen öncelik ondalık (decimal) 95 olarak belirtilebilir. Ondalık 95, ikili 101nnnn (burada 'n' uygulanmamış bir bittir) oluşturmak için dört kaydırılan ve uygulanmamış bitlerin ikili 1011111 oluşturmak için 1'e ayarlandığı ikili 101'dir.

Bazı kütüphane işlevleri, öncelik değerlerinin uygulanan (en anlamlı) bitlere kaydırılmadan önce belirtilmesini bekler. Böyle bir işlev kullanıldığında Şekil 7.15'te gösterilen öncelik ondalık 5 olarak belirtilmelidir. Ondalık 5, herhangi bir kaydırma (shift) olmaksızın ikili 101'dir.

`configMAX_SYSCALL_INTERRUPT_PRIORITY` ve `configKERNEL_INTERRUPT_PRIORITY` doğrudan Cortex-M yazmaçlarına (registers) yazılabilmelerine olanak tanıyacak şekilde, yani öncelik değerleri uygulanan bitlere kaydırıldıktan sonra belirtilmelidir.

`configKERNEL_INTERRUPT_PRIORITY` her zaman olası en düşük kesme önceliğine ayarlanmalıdır. Uygulanmayan öncelik bitleri 1 olarak ayarlanabilir, bu nedenle gerçekte kaç öncelik bitinin uygulandığına bakılmaksızın sabit (constant) her zaman 255'e ayarlanabilir.

Cortex-M kesmeleri (interrupts) varsayılan (default) olarak sıfır önceliğine sahip olacaktır - olası en yüksek öncelik. Cortex-M donanımının uygulanması `configMAX_SYSCALL_INTERRUPT_PRIORITY` sabitinin 0'a ayarlanmasına izin vermez, bu nedenle FreeRTOS API'sini kullanan bir kesmenin önceliği asla varsayılan değerinde bırakılmamalıdır.

8 Kaynak Yönetimi (Resource Management)

8.1 Giriş

Çok görevli (multitasking) bir sistemde, bir görev bir kaynağa (resource) erişmeye başlar, ancak Çalışıyor (Running) durumundan çıkarılmadan önce erişimini tamamlamazsa hata potansiyeli vardır. Eğer görev kaynağı tutarsız bir durumda bırakırsa, aynı kaynağa başka herhangi bir görev veya kesme (interrupt) tarafından erişilmesi veri bozulmasına (data corruption) veya diğer benzer sorunlara neden olabilir.

Aşağıda bazı örnekler verilmiştir:

- **Çevre Birimlerine Erişim (Accessing Peripherals):** İki görevin bir Sıvı Kristal Ekrana (LCD) yazmaya çalıştığı aşağıdaki senaryoyu düşünün.
    1. Görev A yürütülür ve LCD'ye "Hello world" dizesini yazmaya başlar.
    2. Görev A, dizenin sadece başını—"Hello w"—çıkardıktan sonra Görev B tarafından engellenir (pre-empted).
    3. Görev B, Engellenmiş (Blocked) duruma girmeden önce LCD'ye "Abort, Retry, Fail?" yazar.
    4. Görev A, engellendiği noktadan devam eder ve dizesinin kalan karakterlerini—"orld"—çıkarmayı tamamlar.

  LCD şimdi bozuk (corrupted) dizeyi gösterir: "Hello wAbort, Retry, Fail?orld".

- **Oku, Değiştir, Yaz İşlemleri (Read, Modify, Write Operations):** Liste 8.1 bir C kodu satırını ve bu C kodunun assembly koduna tipik olarak nasıl çevrileceğine dair bir örneği göstermektedir. `PORTA` değerinin önce bellekten bir yazmaca (register) okunduğu, yazmaç içinde değiştirildiği ve ardından belleğe geri yazıldığı görülebilir. Buna oku, değiştir, yaz işlemi denir.

```
/* Derlenen C kodu. */
PORTA |= 0x01;

/* C kodu derlendiğinde üretilen assembly kodu. */
LOAD R1,[#PORTA] ; PORTA'dan R1'e bir değer okuyun
MOVE R2,#0x01    ; Mutlak sabit 1'i R2'ye taşıyın
```

```
•  OR   R1,R2       ; R1 (PORTA) ile R2'yi (sabit 1) Bitwise (Bitsel) VEYA
   işlemine sokun
```

```
STORE R1,[#PORTA] ; Yeni değeri PORTA'ya geri kaydedin
```

Liste 8.1 Örnek bir oku, değiştir, yaz dizisi

Bu 'atomik olmayan' (non-atomic) bir işlemdir çünkü tamamlanması birden fazla komut gerektirir ve kesintiye uğratılabilir (interrupted). İki görevin `PORTA` adı verilen belleğe eşlenmiş (memory mapped) bir yazmacı güncellemeye çalıştığı aşağıdaki senaryoyu düşünün.

1. Görev A `PORTA` değerini bir yazmaca (register) yükler; bu, işlemin okuma (read) kısmıdır.
2. Görev A, aynı işlemin değiştirme (modify) ve yazma (write) kısımlarını tamamlamadan önce Görev B tarafından engellenir (pre-empted).
3. Görev B `PORTA`'nın değerini günceller, ardından Engellenmiş durumuna girer.
4. Görev A, engellendiği noktadan itibaren devam eder. Güncellenmiş değeri `PORTA`'ya geri yazmadan önce, halihazırda bir yazmaçta tuttuğu `PORTA` değerinin kopyasını değiştirir.

Bu senaryoda Görev A, `PORTA` için güncel olmayan bir değeri günceller ve geri yazar. Görev B, Görev A, `PORTA` değerinin bir kopyasını aldıktan sonra ve Görev A değiştirilmiş değerini `PORTA` yazmacına geri yazmadan önce `PORTA`'yı değiştirir. Görev A, `PORTA`'ya yazdığında, Görev B tarafından önceden gerçekleştirilen değişikliğin üzerine yazar ve `PORTA` yazmacının değerini fiilen bozar (corrupts).

Bu örnekte bir çevresel yazmaç (peripheral register) kullanılmıştır, ancak değişkenler (variables) üzerinde oku, değiştir, yaz işlemleri gerçekleştirilirken de aynı ilke geçerlidir.

- **Değişkenlere Atomik Olmayan Erişim (Non-atomic Access to Variables):** Bir yapının (structure) birden çok üyesinin (member) güncellenmesi veya mimarinin (architecture) doğal sözcük boyutundan (natural word size) daha büyük bir değişkenin güncellenmesi (örneğin, 16 bitlik bir makinede 32 bitlik bir değişkenin güncellenmesi), atomik olmayan işlemlere örnektir. Kesintiye uğrarlarsa (interrupted), veri kaybına veya bozulmasına neden olabilirler.
- **İşlevlerin Yeniden Girişi (Function Reentrancy):** Eğer bir işlev, kendi çağrılması arasında herhangi bir görevden veya ISR'den, durumu ve verileri bozulmadan güvenli bir şekilde çağrılabiliyorsa, bu işlevin (fonksiyonun) 'yeniden girilebilir' (reentrant) olduğu söylenir. Başka bir deyişle, yeniden girilebilir bir işlevin aynı anda (concurrently) çalıştırılan birden fazla bağlamı (context) olabilir.

Bölüm 3, her görevin yığın (stack) olarak kullanılmak üzere tahsis edilmiş

kendi bellek bloğunu (block of memory) nasıl koruduğunu açıklamıştır. Her görev, yığında (stack) tahsis edilen yerel değişkenlerin (local variables) kendi kopyasına ve o anda C fonksiyonunda yürütülmekte olan kod tarafından kullanılan yığın çerçevesinin (stack frame) kendi kopyasına sahiptir. Yalnızca yığın üzerinde tahsis edilmiş değişkenlere veya bir görevin yazmaçlarında tutulan değişkenlere erişen bir C fonksiyonu, doğası gereği yeniden girilebilirdir (reentrant).

Sadece erişildiği görev tarafından sahip olunmayan verilere erişen bir fonksiyon, yeniden girilebilir değildir (non-reentrant). Doğası gereği aynı anda birden fazla görev tarafından erişilmekte olan bir donanım çevre birimine (hardware peripheral) erişen bir fonksiyon gibi.

8.1.1 Karşılıklı Dışlama (Mutual Exclusion)

Paylaşılan bir kaynağın (shared resource) hiçbir zaman bozulmamasını (corrupted) sağlamak için, sistem tarafından kaynağa erişimi denetlemek üzere "karşılıklı dışlama" (mutual exclusion) adı verilen işlemler kullanılmalıdır. Bunlar tekniklerdir. Karşılıklı dışlama yoluyla sağlanan veri tutarlılığı her zaman gerekli değildir. Örneğin, bir uygulamayı tasarlarken, birden fazla görevin ortak bir veri yapısına, veriler yalnızca salt okunur (read only) olacak şekilde veya bir görevin değişkenin tamamını, okuma işleminden başka bir görevin de değiştirme riski olmadan güncelleyecek şekilde erişmesi gibi, değişken güncellemelerinin atomik olduğu (atomic variable update) durumları garantilemek mümkündür.

Aynı şekilde, bazen belirli veri öğelerinin bir ISR ile paylaşılması bir sorun olmayabilir. Bir veri öğesine hem ISR'den hem de uygulamadaki başka bir yerden erişiliyorsa, değerin bir kopyası da oluşturulmalı veya değeri test edilmelidir; böylece verilerin bir ISR içindeki değişikliği önlenebilir.

Daha karmaşık sistemler, bir görev (veya ISR), başka bir görev (veya ISR) tarafından zaten paylaşılıyorsa ve kaynağı özel olarak (exclusively) kontrol etmesi gereken bir bölüme ulaşırsa, işleme izin verilmeden önce o görev bitene kadar beklenmesini sağlamalıdır.
Bölümün geri kalanı, FreeRTOS'un karşılıklı dışlamayı uygulamak için sunduğu mekanizmaları anlatmaktadır.

8.1.2 Kapsam

Bu bölüm şunları kapsamaktadır:

- Kritik bölümler (critical sections) ve FreeRTOS'ta uygulanan temel kullanım alanları (basic use cases).
- Karşılıklı Dışlama (Mutexes) ve ikili semaforlar arasındaki fark.
- Kritik bölümler ve Karşılıklı Dışlama (Mutex) mekanizmaları ne zaman ve nasıl kullanılır.
- Öncelik Ters Çevirme (Priority inversion).
- Öncelik Mirası (Priority inheritance).

8.2 Kritik Bölümler ve Çizgeleyicinin Askıya Alınması (Critical Sections and Suspending the Scheduler)

8.2.1 Temel Kritik Bölümler (Basic Critical Sections)

Temel (basic) kritik bölümler, sırasıyla `taskENTER_CRITICAL()` ve `taskEXIT_CRITICAL()` makrolarıyla çevrelenen kod bölgeleridir. Kritik bölümler aynı zamanda kritik bölgeler (critical regions) olarak da bilinir.

`taskENTER_CRITICAL()` ve `taskEXIT_CRITICAL()` işlevleri bir makro sağlarlar, dolayısıyla dönüş (return) ve parametre (parameter) gerektirmezler. Liste 8.5, her ikisi de bir FreeRTOS uyarlamasının bir makrosu olan `vPrintString()` öğesinin nasıl yapılandırıldığını (how to form a block) gösteren bu mekanizmayı kullanır.

```
void vPrintString( const char *pcString )

{

    /* Dizeyi (string) standart çıktıya (standard out) yazın. Terminale yazdırma

       işlemi bitene kadar başka hiçbir görevin veya ISR'nin çalışamayacağından,

       yani stdout'a aynı anda yalnızca bir görevin erişebileceğinden (mutually

       exclusive access) emin olmak için kritik bir bölüm (critical section)
kullanılır. */

    taskENTER_CRITICAL();

    {

        printf( "%s", pcString );

        fflush( stdout );

    }

    taskEXIT_CRITICAL();

}
```

Liste 8.5 `vPrintString()` işlevinin olası bir uygulaması (implementation)

Bu şekilde uygulanan kritik bölümler karşılıklı dışlama sağlamanın çok kaba (crude) bir yöntemidir. Kullanılan FreeRTOS portuna bağlı olarak, kesmeleri tamamen ya da `configMAX_SYSCALL_INTERRUPT_PRIORITY` tarafından ayarlanan kesme önceliğine (interrupt priority) kadar devre dışı bırakarak (disabling) çalışırlar. Engelleyici (pre-emptive) bağlam geçişleri (context switches) yalnızca bir kesme içinden gerçekleşebilir, bu nedenle, kesmeler devre dışı kaldığı sürece, `taskENTER_CRITICAL()`

çağrısı yapan görevin kritik bölümden çıkılana kadar Çalışıyor durumunda (Running state) kalması garanti edilir.

Temel kritik bölümler çok kısa tutulmalıdır, aksi takdirde kesme yanıt sürelerini (interrupt response times) olumsuz etkilerler. `taskENTER_CRITICAL()` çağrısı, her seferinde bir `taskEXIT_CRITICAL()` çağrısıyla yakından eşleştirilmelidir. Bu nedenle standart çıktı (stdout veya bilgisayarın çıktı verilerini yazdığı akış), terminale yazma işlemi nispeten uzun bir işlem olabileceğinden kritik bir bölüm kullanılarak korunmamalıdır (Liste 8.5'te gösterildiği gibi). Bu bölümdeki örnekler alternatif çözümleri incelemektedir.

Çekirdek yuvalama (nesting) derinliğinin bir sayımını (count) tuttuğundan, kritik bölümlerin iç içe geçmesi (nested) güvenlidir. Kritik bölümden, ancak yuvalama derinliği sıfıra döndüğünde çıkılacaktır; bu da her bir önceki `taskENTER_CRITICAL()` çağrısı için bir `taskEXIT_CRITICAL()` çağrısının yürütülmesi (executed) anlamına gelir.

`taskENTER_CRITICAL()` ve `taskEXIT_CRITICAL()` makrolarını çağırmak, FreeRTOS'un üzerinde çalıştığı işlemcinin (processor) kesme etkinleştirme (interrupt enable) durumunu değiştirmek için bir görev için yegane yasal (legitimate) yoldur. Kesme etkinleştirme durumunu başka bir yolla değiştirmek, makronun yuvalama (nesting) sayısını (count) geçersiz kılacaktır.

`taskENTER_CRITICAL()` ve `taskEXIT_CRITICAL()` işlevleri 'FromISR' ile bitmez, bu nedenle bir kesme servis rutininden çağrılmamalıdır. `taskENTER_CRITICAL_FROM_ISR()`, `taskENTER_CRITICAL()` makrosunun kesme güvenli sürümüdür ve `taskEXIT_CRITICAL_FROM_ISR()`, `taskEXIT_CRITICAL()` makrosunun kesme güvenli sürümüdür. Kesme güvenli (interrupt safe) sürümler yalnızca kesmelerin yuvalanmasına (nest) izin veren FreeRTOS portları için sağlanır; kesmelerin yuvalanmasına izin vermeyen portlarda kullanımdan kalkmış (obsolete) olacaklardır.

`taskENTER_CRITICAL_FROM_ISR()` işlevi, eşleşen `taskEXIT_CRITICAL_FROM_ISR()` çağrısına geçirilmesi (passed) gereken bir değer döndürür. Bu durum Liste 8.6'da gösterilmektedir.

Çizgeleyici `vTaskSuspendAll()` çağrılarak askıya alınır. Çizgeleyicinin askıya alınması, bir bağlam geçişinin (context switch) meydana gelmesini engeller, ancak kesmeleri (interrupts) etkin bırakır. Çizgeleyici askıya alınmışken bir kesme bir bağlam geçişi talep ederse, istek beklemede (pending) tutulur ve yalnızca çizgeleyici devam ettirildiğinde (askıya alınma durumu kaldırıldığında) gerçekleştirilir. Çizgeleyici askıya alınmışken FreeRTOS API işlevleri çağrılmamalıdır.

8.2.4 xTaskResumeAll() API İşlevi

```
BaseType_t xTaskResumeAll( void );
```

Liste 8.8 `xTaskResumeAll()` API işlevi prototipi

Çizgeleyici `xTaskResumeAll()` çağrılarak devam ettirilir (askıya alma durumu kaldırılır).

`xTaskResumeAll()` dönüş değeri:

- **Dönüş değeri (Return value):** Çizgeleyici askıya alınmışken talep edilen bağlam geçişleri beklemede (pending) tutulur ve yalnızca çizgeleyici devam ettirilirken gerçekleştirilir. Bekleyen bir bağlam geçişi `xTaskResumeAll()` işlevi dönmeden (returns) önce gerçekleştirilirse, o zaman `pdTRUE` döndürülür. Aksi takdirde `pdFALSE` döndürülür.

`vTaskSuspendAll()` ve `xTaskResumeAll()` çağrılarının iç içe geçmesi (nested) güvenlidir, çünkü çekirdek yuvalama derinliğinin bir sayımını tutar. Çizgeleyici yalnızca yuvalama derinliği sıfıra döndüğünde devam ettirilir — bu da, her bir önceki `vTaskSuspendAll()` çağrısı için bir `xTaskResumeAll()` çağrısı yürütüldüğü zamandır.

Liste 8.9, terminal çıktısına erişimi korumak için çizgeleyiciyi askıya alan `vPrintString()` işlevinin gerçek uygulamasını göstermektedir.

```
void vPrintString( const char *pcString )
{
    /* Karşılıklı dışlama yöntemi olarak çizgeleyiciyi askıya alıp
       dizeyi stdout'a yazın. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();
}
```

Liste 8.9 `vPrintString()` işlevinin uygulanması

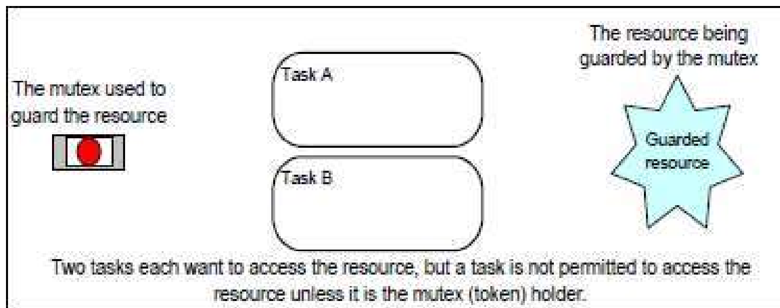
8.3 Karşılıklı Dışlamalar (Mutexes) ve İkili Semaforlar (Binary Semaphores)

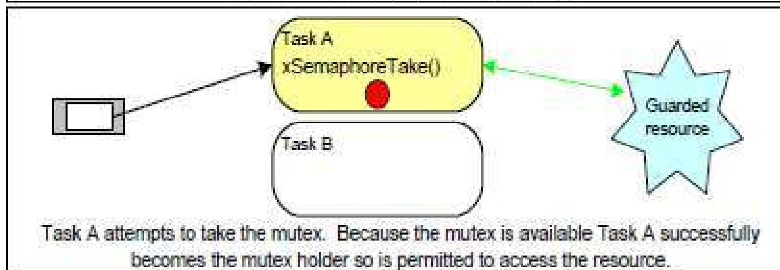
Bir Mutex (Karşılıklı Dışlama), iki veya daha fazla görev arasında paylaşılan bir kaynağa erişimi kontrol etmek için kullanılan özel bir ikili semafor türüdür. MUTEX kelimesi 'MUTual EXclusion' (Karşılıklı Dışlama) kelimesinden gelir. Karşılıklı dışlamaların kullanılabilmesi için `FreeRTOSConfig.h` dosyasında `configUSE_MUTEXES` 1 olarak ayarlanmalıdır.

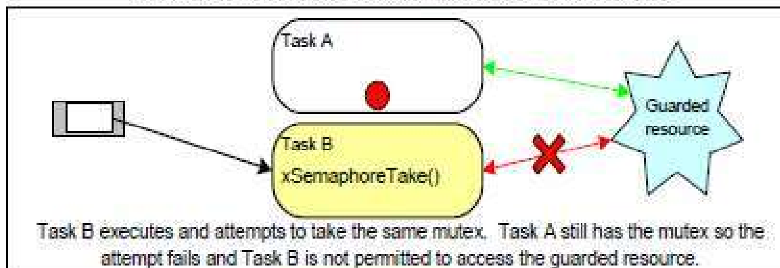
Karşılıklı dışlama senaryosunda kullanıldığında, bir mutex, paylaşılan kaynakla ilişkilendirilmiş bir "jeton" (token) olarak düşünülebilir. Bir görevin kaynağa meşru bir şekilde erişebilmesi için, öncelikle jetonu başarıyla "alması" (take) (jeton sahibi olması) gerekir. Jeton sahibi kaynakla işini bitirdiğinde, jetonu geri "vermelidir" (give). Sadece jeton iade edildiğinde başka bir görev jetonu başarıyla alabilir ve ardından aynı paylaşılan kaynağa güvenle erişebilir. Bir görevin, jetonu elinde bulundurmadığı sürece paylaşılan kaynağa erişmesine izin verilmez. Bu mekanizma Şekil 8.1'de gösterilmektedir.

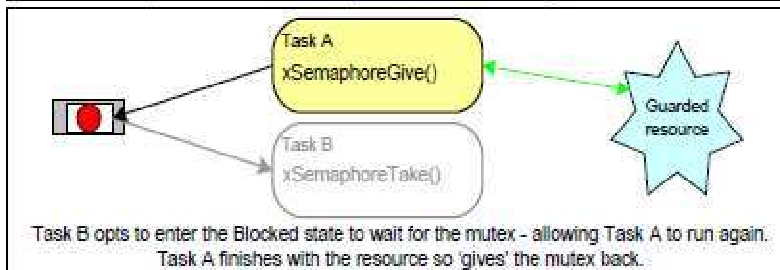
Karşılıklı dışlamalar (mutexes) ve ikili semaforlar birçok ortak özelliği paylaşsa da, Şekil 8.1'de (bir mutex'in karşılıklı dışlama için kullanıldığı yer) gösterilen senaryo, Şekil 7.6'da (bir ikili semaforun senkronizasyon için kullanıldığı yer) gösterilenden tamamen farklıdır. Temel fark, elde edildikten sonra semafora ne olduğudur:

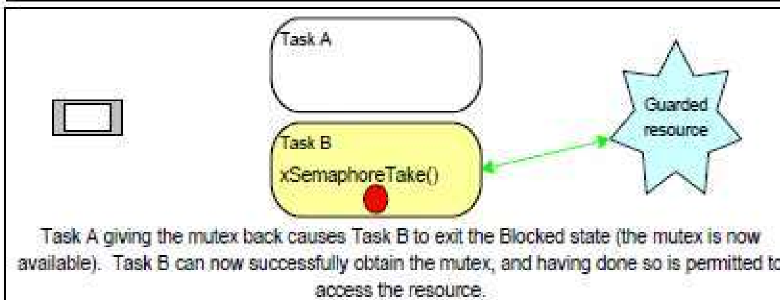
- Karşılıklı dışlama (mutual exclusion) için kullanılan bir semafor **her zaman geri döndürülmelidir (returned)**.
- Senkronizasyon için kullanılan bir semafor ise **normalde atılır (discarded) ve geri döndürülmez**.

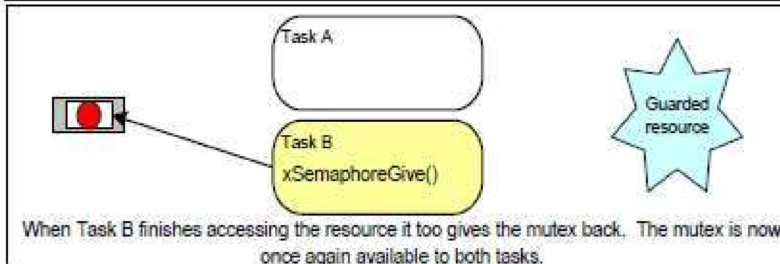

The resource being guarded by the mutex
The mutex used to guard the resource
Task A
Task B
Guarded resource
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.


Task A
xSemaphoreTake()
Task B
Guarded resource
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.


Task A
Task B
xSemaphoreTake()
Guarded resource
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.


Task A
xSemaphoreGive()
Task B
xSemaphoreTake()
Guarded resource
Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.


Task A
Task B
xSemaphoreTake()
Guarded resource
Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.


Task A
Task B
xSemaphoreGive()
Guarded resource
When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

Şekil 8.1 Bir mutex kullanılarak uygulanan karşılıklı dışlama

Mekanizma tamamen uygulama yazarının (application writer) disiplini ile çalışır. Bir görevin herhangi bir zamanda kaynağa erişememesi için hiçbir neden yoktur, ancak her bir görev, mutex sahibi olamadığı sürece erişmemeyi "kabul eder".

8.3.1 xSemaphoreCreateMutex() API İşlevi

FreeRTOS ayrıca derleme zamanında (compile time) statik olarak bir mutex oluşturmak için gereken belleği tahsis eden (allocates) `xSemaphoreCreateMutexStatic()` işlevini de içerir: Bir mutex, bir semafor türüdür. Tüm çeşitli FreeRTOS semafor türlerine ait tanıtıcılar (handles), `SemaphoreHandle_t` türünde bir değişkende saklanır.

Bir mutex kullanılabilmesi için önce oluşturulması gerekir. Mutex türü bir semafor oluşturmak için `xSemaphoreCreateMutex()` API işlevini kullanın.

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Liste 8.10 `xSemaphoreCreateMutex()` API işlevi prototipi

`xSemaphoreCreateMutex()` dönüş değeri:

- **Dönüş değeri:** Eğer `NULL` döndürülürse, mutex oluşturulamamıştır çünkü FreeRTOS'un mutex veri yapılarını tahsis etmesi (allocate) için mevcut yeterli yığın (heap) belleği yoktur. Bölüm 3 yığın belleği yönetimi hakkında daha fazla bilgi sağlamaktadır.

Örnek 8.1 `vPrintString()` işlevini bir semafor kullanacak şekilde yeniden yazma

Bu örnek, `vPrintString()` işlevinin `prvNewPrintString()` adında yeni bir sürümünü oluşturur, ardından bu yeni işlevi birden çok görevden çağırır. `prvNewPrintString()` işlevsel olarak `vPrintString()` ile aynıdır, ancak standart çıkışa (standard out) erişimi çizgeleyiciyi kilitlemek yerine bir mutex kullanarak kontrol eder. `prvNewPrintString()` uygulaması Liste 8.11'de gösterilmektedir.

```
static void prvNewPrintString( const char *pcString )

{

    /* Mutex çizgeleyici başlatılmadan önce oluşturulur, bu nedenle

       bu görev yürütüldüğünde zaten mevcuttur.
```

```
        Mutex'i almayı (take) deneyin, eğer hemen mevcut değilse mutex
        için süresiz olarak engellenin (bekleyin). xSemaphoreTake() çağrısı
        yalnızca mutex başarıyla elde edildiğinde dönecektir, bu nedenle
        işlevin dönüş değerini kontrol etmeye gerek yoktur. Başka bir
        gecikme süresi kullanılmış olsaydı, paylaşılan kaynağa (bu durumda
        standart çıktı - standard out) erişmeden önce xSemaphoreTake()
        işlevinin pdTRUE döndürdüğünü kontrol etmek gerekirdi. Bu kitapta
        daha önce belirtildiği gibi, üretim (production) kodu için süresiz
        zaman aşımları (indefinite time outs) önerilmez. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* Aşağıdaki satır yalnızca mutex başarıyla elde edildiğinde
           yürütülecektir. Herhangi bir anda mutex'e yalnızca bir görev
           sahip olabileceğinden artık standart çıktıya serbestçe erişilebilir. */
        printf( "%s", pcString );
        fflush( stdout );

        /* Mutex GERİ VERİLMELİDİR! (MUST be given back!) */
    }
    xSemaphoreGive( xMutex );
}
```

Liste 8.11 `prvNewPrintString()` işlevinin uygulanması

`prvNewPrintString()` işlevi, `prvPrintTask()` tarafından uygulanan bir görevin iki örneği (instance) tarafından art arda (repeatedly) çağrılır. Her çağrı arasında rastgele (random) bir gecikme süresi kullanılır. Görev parametresi (task parameter), görevin her bir örneğine benzersiz bir dize (unique string) geçirmek için kullanılır. `prvPrintTask()` uygulaması Liste 8.12'de gösterilmektedir.

```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;
```

```
    const TickType_t xMaxBlockTimeTicks = 0x20;


    /* Bu görevin iki örneği oluşturulur. Görev tarafından yazdırılan dize,

       görevin parametresi kullanılarak göreve geçirilir. Parametre,

       gerekli türe (type) dönüştürülür (cast). */

    pcStringToPrint = ( char * ) pvParameters;


    for( ;; )

    {

        /* Yeni tanımlanan işlevi kullanarak dizeyi yazdırın. */

        prvNewPrintString( pcStringToPrint );


        /* Sözde rastgele (pseudo random) bir süre bekleyin. rand() işlevinin

           ilgisiz (necessarily) olarak yeniden girilebilir (reentrant) olmadığını

           unutmayın, ancak bu durumda kodun hangi değerin döndürüldüğünü

           önemsememesi nedeniyle bunun pek bir önemi yoktur. Daha güvenli

           bir uygulamada, rand() işlevinin yeniden girilebilir olduğu bilinen bir

           sürümü kullanılmalı veya rand() çağrıları kritik bir bölüm (critical

           section) kullanılarak korunmalıdır. */

        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );

    }

}
```

Liste 8.12 Örnek 8.1 için `prvPrintTask()` işlevinin uygulanması

Normalde olduğu gibi `main()`, mutex'i oluşturur, görevleri oluşturur, ardından çizgeleyiciyi başlatır. Uygulama Liste 8.13'te gösterilmektedir.

`prvPrintTask()` işlevinin iki örneği farklı önceliklerde oluşturulur, bu nedenle daha düşük öncelikli görev bazen daha yüksek öncelikli görev tarafından engellenir (pre-empted). Her görevin terminale karşılıklı olarak özel (mutually exclusive) erişim elde etmesini sağlamak için bir mutex kullanıldığından, engelleme gerçekleştiğinde bile görüntülenen dizeler (strings) doğru olacak ve hiçbir şekilde bozulmayacaktır (corrupted). Engelleme sıklığı, `xMaxBlockTimeTicks` sabiti tarafından ayarlanan,

görevlerin Engellenmiş durumunda geçirdiği maksimum sürenin azaltılmasıyla artırılabilir.

Örnek 8.1'in FreeRTOS Windows portu ile kullanımına özgü notlar:

- `printf()` işlevini çağırmak bir Windows sistem çağrısı (system call) üretir. Windows sistem çağrıları FreeRTOS'un kontrolü dışındadır ve kararsızlığa (instability) neden olabilir.
- Windows sistem çağrılarının yürütülme şekli, mutex kullanılmadığında bile bozuk bir dize görmenin nadir olduğu anlamına gelir.

```
int main( void )

{

    /* Bir semafor kullanılmadan önce açıkça oluşturulmalıdır. Bu örnekte

       mutex türü bir semafor oluşturulmuştur. */

    xMutex = xSemaphoreCreateMutex();


    /* Görevleri oluşturmadan önce semaforun başarıyla oluşturulduğunu

       kontrol edin. */

    if( xMutex != NULL )

    {

        /* stdout'a yazan görevlerin iki örneğini (instance) oluşturun.

           Yazdıkları dize, görevin parametresi olarak göreve geçirilir.

           Görevler farklı önceliklerde oluşturulur, bu nedenle bir miktar

           engelleme (pre-emption) meydana gelecektir. */

        xTaskCreate( prvPrintTask, "Print1", 1000,

                     "Task 1 ***************************************\r\n",

                     1, NULL );


        xTaskCreate( prvPrintTask, "Print2", 1000,

                     "Task 2 ---------------------------------------\r\n",

                     2, NULL );
```

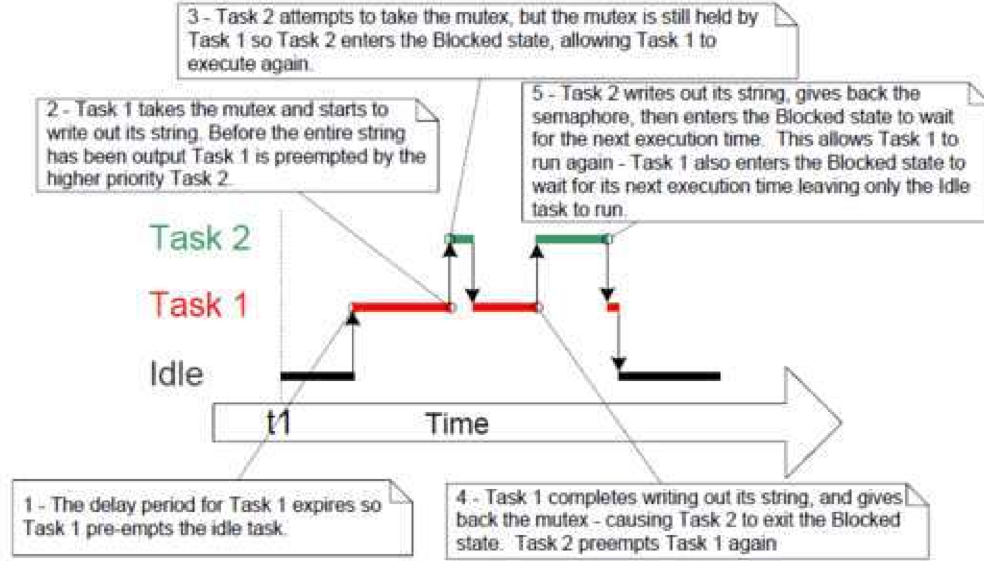
```
        /* Oluşturulan görevlerin çalışmaya başlaması için çizgeleyiciyi başlatın.
*/

        vTaskStartScheduler();

    }


    /* Her şey yolundaysa çizgeleyici artık görevleri çalıştıracağından

       main() buraya asla ulaşmayacaktır. main() buraya ulaşırsa,

       boşta kalan görevin (idle task) oluşturulması için yeterli

       yığın (heap) belleği bulunmamış olması muhtemeldir. Bölüm 3

       yığın belleği yönetimi hakkında daha fazla bilgi sağlar. */

    for( ;; );

}
```

Liste 8.13 Örnek 8.1 için `main()` işlevinin uygulanması

Örnek 8.1 çalıştırıldığında üretilen çıktı Şekil 8.2'de gösterilmektedir. Olası bir yürütme dizisi Şekil 8.3'te açıklanmaktadır.

Şekil 8.2 Örnek 8.1 çalıştırıldığında üretilen çıktı

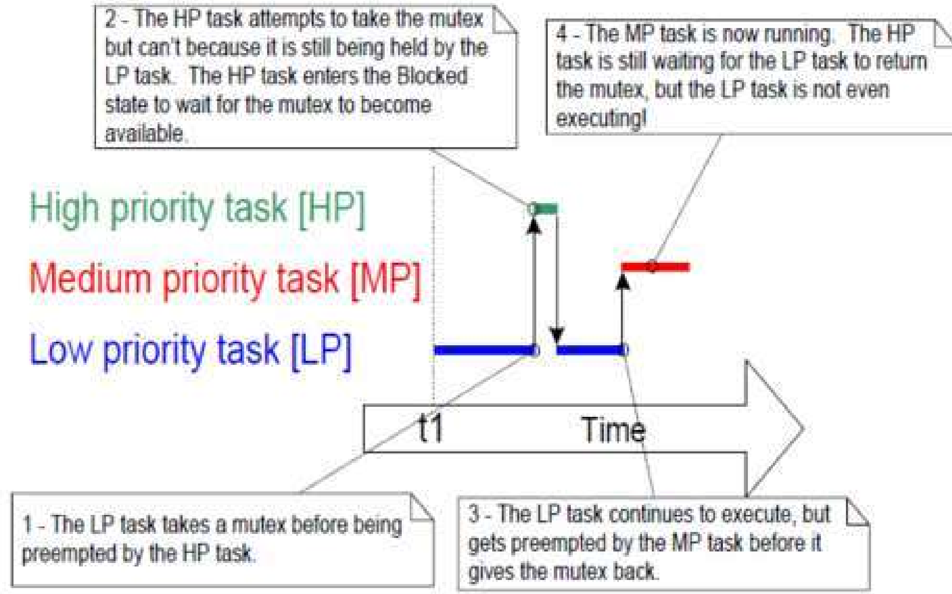
Şekil 8.2 beklendiği gibi terminalde görüntülenen dizelerde herhangi bir bozulma (corruption) olmadığını göstermektedir. Rastgele sıralama, görevler tarafından kullanılan rastgele gecikme sürelerinin (random delay periods) bir sonucudur.



Şekil 8.3 Örnek 8.1 için olası bir yürütme dizisi (sequence of execution)

8.3.2 Öncelik Ters Çevirme (Priority Inversion)

Şekil 8.3, karşılıklı dışlama sağlamak için bir mutex kullanmanın olası tuzaklarından (pitfalls) birini göstermektedir. Tasvir edilen yürütme dizisi (sequence of execution), daha yüksek öncelikli Görev 2'nin, daha düşük öncelikli Görev 1'in mutex'in kontrolünü bırakmasını beklemek zorunda olduğunu göstermektedir. Daha yüksek öncelikli bir görevin daha düşük öncelikli bir görev tarafından bu şekilde geciktirilmesine "öncelik ters çevirme" (priority inversion) adı verilir. Yüksek öncelikli görev semaforu beklerken orta öncelikli bir görev yürütülmeye başlasaydı, bu istenmeyen davranış daha da abartılırdı — sonuç, yüksek öncelikli bir görevin düşük öncelikli bir görevi beklemesi olurdu — düşük öncelikli görev yürütülemese bile. Bu en kötü durum (worst case) senaryosu Şekil 8.4'te gösterilmektedir.



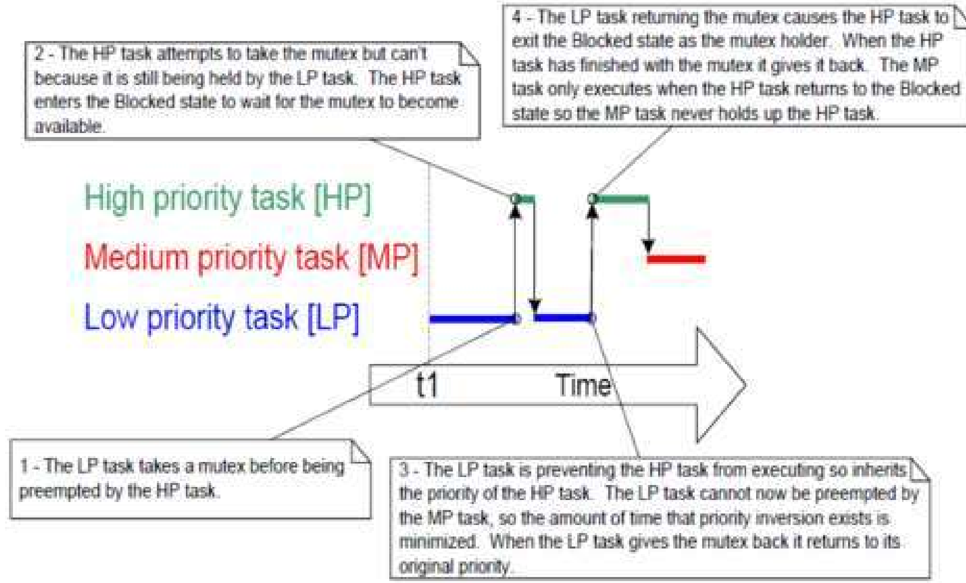
Şekil 8.4 En kötü senaryoda öncelik ters çevirme durumu

Öncelik ters çevirme önemli bir sorun olabilir, ancak küçük gömülü sistemlerde (embedded systems) kaynaklara nasıl erişileceği göz önünde bulundurularak, sistem tasarım aşamasında genellikle bundan kaçınılabilir.

8.3.3 Öncelik Mirası (Priority Inheritance)

FreeRTOS mutex'leri ve ikili semaforları birbirine çok benzer — aralarındaki fark, mutex'lerin temel bir "öncelik mirası" (priority inheritance) mekanizması içermesi, ikili semaforların ise içermemesidir. Öncelik mirası, öncelik ters çevirmenin (priority inversion) olumsuz etkilerini en aza indiren bir şemadır. Öncelik ters çevirmeyi "düzeltmez" (fix), ancak ters çevirmenin her zaman zamanla sınırlı (time bounded) olmasını sağlayarak etkisini sadece azaltır. Ancak öncelik mirası sistem zamanlama (timing) analizini zorlaştırır ve sistemin doğru çalışması için buna güvenmek iyi bir uygulama değildir.

Öncelik mirası, mutex sahibinin önceliğinin geçici olarak aynı mutex'i elde etmeye çalışan en yüksek öncelikli görevin önceliğine yükseltilmesiyle çalışır. Mutex'i elinde tutan düşük öncelikli görev, mutex'i bekleyen görevin önceliğini "devralır" (inherits). Bu durum Şekil 8.5'te gösterilmektedir. Mutex sahibi mutex'i geri verdiğinde önceliği otomatik olarak orijinal değerine sıfırlanır.



Şekil 8.5 Öncelik ters çevirmenin etkisini en aza indiren öncelik mirası

Az önce görüldüğü gibi, öncelik mirası işlevselliği, mutex'i kullanan görevlerin önceliğini etkiler. Bu nedenle mutex'ler kesme servis rutinlerinden (interrupt service routines) kullanılmamalıdır.

8.3.4 Kilitlenme (Deadlock veya Deadly Embrace)

'Kilitlenme' (Deadlock), karşılıklı dışlama için mutex kullanmanın bir başka potansiyel tuzağıdır (pitfall). Kilitlenme bazen 'ölümcül kucaklaşma' (deadly embrace) gibi daha dramatik bir isimle de bilinir.

Kilitlenme, iki görevin devam edemediği durumlarda meydana gelir, çünkü her ikisi de diğerinin elinde tuttuğu bir kaynağı bekliyordur. Görev A ve Görev B'nin bir eylemi gerçekleştirmek için hem X mutex'ini hem de Y mutex'ini elde etmesi (acquire) gereken aşağıdaki senaryoyu düşünün:

1. Görev A yürütülür ve X mutex'ini başarıyla alır.
2. Görev A, Görev B tarafından engellenir (pre-empted).
3. Görev B aynı zamanda X mutex'ini almaya çalışmadan önce Y mutex'ini başarıyla alır — ancak X mutex'i Görev A tarafından tutulmaktadır, bu nedenle Görev B için mevcut değildir. Görev B, X mutex'inin serbest bırakılmasını (released) beklemek için Engellenmiş durumuna girmeyi seçer.
4. Görev A yürütülmeye devam eder. Y mutex'ini almaya çalışır — ancak Y mutex'i Görev B tarafından tutulmaktadır, bu nedenle Görev A için mevcut değildir. Görev A, Y mutex'inin serbest bırakılmasını beklemek için Engellenmiş durumuna girmeyi seçer.

Bu senaryonun sonunda Görev A, Görev B tarafından tutulan bir mutex'i beklemektedir ve Görev B de Görev A tarafından tutulan bir mutex'i beklemektedir. Her iki görev de ilerleyemediği için kilitlenme (deadlock) meydana gelmiştir.

Öncelik ters çevirmede (priority inversion) olduğu gibi, kilitlenmeden kaçınmanın en iyi yöntemi, tasarım zamanında (design time) potansiyelini göz önünde bulundurmak ve sistemin kilitlenme meydana gelemeyeceğinden emin olacak şekilde tasarlanmasıdır. Özellikle ve bu kitapta daha önce belirtildiği gibi, bir görevin bir mutex'i elde etmek için süresiz olarak (bir zaman aşımı - time out olmadan) beklemesi normalde kötü bir uygulamadır (bad practice). Bunun yerine, mutex'i beklemek için gereken maksimum süreden biraz daha uzun bir zaman aşımı kullanın — o zaman bu süre içinde mutex'i elde edememek, bir kilitlenme olabilecek bir tasarım hatasının belirtisi (symptom) olacaktır.

Pratikte, küçük gömülü sistemlerde (embedded systems) kilitlenme büyük bir sorun değildir, çünkü sistem tasarımcıları tüm uygulamayı iyi anlayabilir ve böylece kilitlenmenin meydana gelebileceği alanları belirleyip ortadan kaldırabilirler.

8.3.5 Özyinelemeli Karşılıklı Dışlamalar (Recursive Mutexes)

Bir görevin kendisiyle kilitlenmesi (deadlock) de mümkündür. Bu durum, bir görev aynı mutex'i önce iade etmeden birden fazla kez almaya çalışırsa meydana gelir. Aşağıdaki senaryoyu düşünün:

1. Bir görev bir mutex'i başarıyla elde eder.
2. Görev, mutex'i elinde tutarken bir kütüphane işlevi (library function) çağırır.
3. Kütüphane işlevinin uygulaması (implementation) aynı mutex'i almaya çalışır ve mutex'in kullanılabilir olmasını beklemek için Engellenmiş durumuna girer.

Bu senaryonun sonunda görev, mutex'in geri dönmesini beklemek üzere Engellenmiş durumundadır, ancak görev zaten mutex'in sahibidir. Görev, kendisini beklemek için Engellenmiş durumunda olduğundan bir kilitlenme meydana gelmiştir.

Bu tür bir kilitlenme, standart bir mutex yerine özyinelemeli (recursive) bir mutex kullanılarak önlenebilir. Özyinelemeli bir mutex, aynı görev tarafından birden fazla kez "alınabilir" (taken) ve yalnızca her bir önceki özyinelemeli mutex "alma" çağrısı için bir özyinelemeli mutex "verme" (give) çağrısı yürütüldükten sonra geri döndürülür.

Standart mutex'ler ve özyinelemeli mutex'ler benzer bir şekilde oluşturulur ve kullanılır:

- Standart mutex'ler `xSemaphoreCreateMutex()` kullanılarak oluşturulur. Özyinelemeli mutex'ler `xSemaphoreCreateRecursiveMutex()` kullanılarak oluşturulur. İki API işlevi aynı prototipe (prototype) sahiptir.
- Standart mutex'ler `xSemaphoreTake()` kullanılarak "alınır". Özyinelemeli mutex'ler `xSemaphoreTakeRecursive()` kullanılarak "alınır". İki API işlevi aynı prototipe sahiptir.

- Standart mutex'ler `xSemaphoreGive()` kullanılarak "verilir". Özyinelemeli mutex'ler `xSemaphoreGiveRecursive()` kullanılarak "verilir". İki API işlevi aynı prototipe sahiptir.

Liste 8.14, özyinelemeli bir mutex'in nasıl oluşturulacağını ve kullanılacağını göstermektedir.

```
/* Özyinelemeli mutex'ler SemaphoreHandle_t türünde değişkenlerdir. */
SemaphoreHandle_t xRecursiveMutex;

/* Özyinelemeli bir mutex oluşturan ve kullanan bir görevin uygulanması. */
void vTaskFunction( void *pvParameters )
{
    const TickType_t xMaxBlock20ms = pdMS_TO_TICKS( 20 );

    /* Özyinelemeli bir mutex kullanılmadan önce açıkça oluşturulmalıdır. */
    xRecursiveMutex = xSemaphoreCreateRecursiveMutex();

    /* Semaforun başarıyla oluşturulduğunu kontrol edin. configASSERT(),
       bölüm 11.2'de açıklanmıştır. */
    configASSERT( xRecursiveMutex );

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngü olarak uygulanır. */
    for( ;; )
    {
        /* ... */

        /* Özyinelemeli mutex'i alın. */
        if( xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms ) == pdPASS )
        {
            /* Özyinelemeli mutex başarıyla elde edildi. Görev artık mutex'in
               koruduğu kaynağa erişebilir. Bu noktada özyinelemeli çağrı
```

```
        sayısı (nested xSemaphoreTakeRecursive() çağrılarının sayısıdır)
        1'dir, çünkü özyinelemeli mutex yalnızca bir kez alınmıştır. */

    /* Görev, halihazırda özyinelemeli mutex'e sahipken, mutex'i tekrar
alır.
        Gerçek bir uygulamada, bunun yalnızca bu görev tarafından çağrılan
        bir alt işlev (sub-function) içinde gerçekleşmesi muhtemeldir,
        çünkü bilerek aynı mutex'i birden fazla kez almak için pratik
        bir neden yoktur. Çağrıyı yapan görev zaten mutex'in sahibidir,
        bu nedenle xSemaphoreTakeRecursive() için yapılan ikinci çağrı
        özyinelemeli çağrı sayısını 2'ye çıkarmaktan başka bir şey yapmaz.
*/
    xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms );

    /* ... */

    /* Görev, mutex'in koruduğu kaynağa erişmeyi bitirdikten sonra
        mutex'i iade eder. Bu noktada özyinelemeli çağrı sayısı 2'dir,
        bu nedenle ilk xSemaphoreGiveRecursive() çağrısı mutex'i
        iade etmez. Bunun yerine, sadece özyinelemeli çağrı sayısını
        1'e düşürür. */
    xSemaphoreGiveRecursive( xRecursiveMutex );

    /* Bir sonraki xSemaphoreGiveRecursive() çağrısı özyinelemeli çağrı
        sayısını 0'a düşürür, bu nedenle bu sefer özyinelemeli mutex
        iade edilir. */
    xSemaphoreGiveRecursive( xRecursiveMutex );

    /* Artık xSemaphoreTakeRecursive() için yapılan her bir işlem
çağrısına
        karşılık bir xSemaphoreGiveRecursive() çağrısı yürütüldüğü için,
```

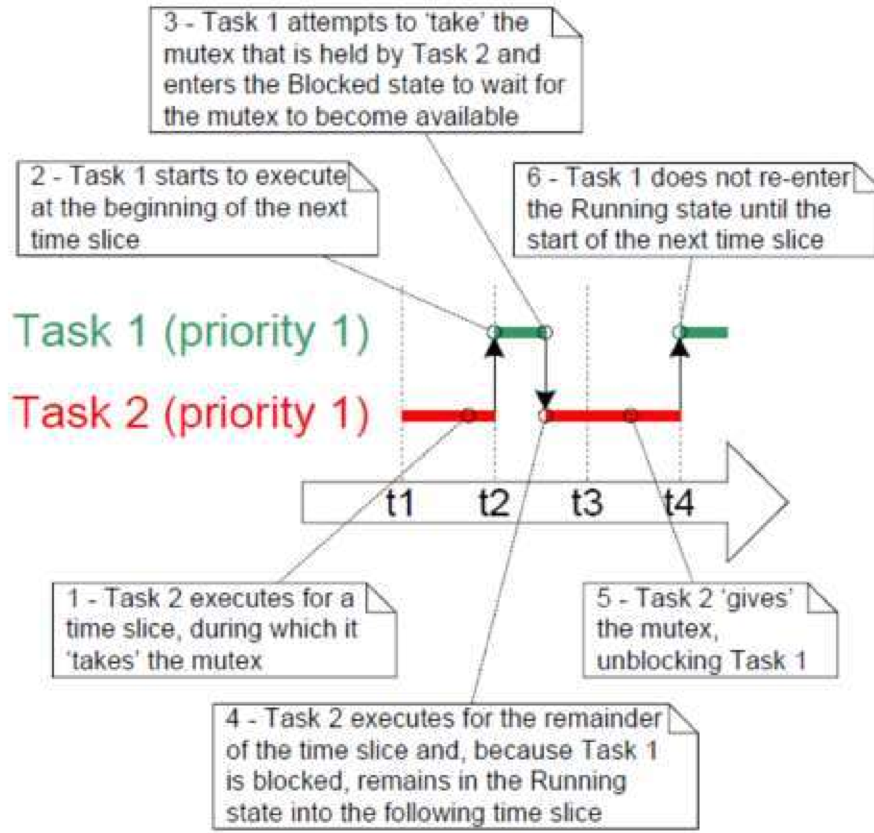
```
                görev artık mutex'in sahibi değildir. */

        }

    }

}
```

Liste 8.14 Özyinelemeli bir mutex oluşturma ve kullanma

8.3.6 Mutex'ler ve Görev Çizgeleme (Mutexes and Task Scheduling)

Farklı önceliklere sahip iki görev aynı mutex'i kullanıyorsa, FreeRTOS çizgeleme ilkesi (scheduling policy) görevlerin hangi sırayla yürütüleceğini (execute) netleştirir; çalışabilen (able to run) en yüksek öncelikli görev, Çalışıyor (Running) durumuna giren görev olarak seçilecektir. Örneğin, yüksek öncelikli bir görev düşük öncelikli bir görev tarafından tutulan bir mutex'i beklemek için Engellenmiş (Blocked) durumundaysa, o zaman yüksek öncelikli görev, düşük öncelikli görev mutex'i iade eder etmez düşük öncelikli görevi engelleyecektir (pre-empt). Ardından yüksek öncelikli görev mutex'in sahibi olacaktır. Bu senaryo zaten Şekil 8.5'te görülmüştür.

Bununla birlikte, görevler aynı önceliğe sahip olduğunda görevlerin yürütülme sırasına ilişkin yanlış bir varsayımda bulunmak yaygındır. Görev 1 ve Görev 2 aynı önceliğe sahipse ve Görev 1, Görev 2 tarafından tutulan bir mutex'i beklemek için Engellenmiş durumundaysa, Görev 2 mutex'i "verdiğinde" (gives) Görev 1, Görev 2'yi engellemeyecektir (will not pre-empt). Bunun yerine, Görev 2 Çalışıyor durumunda kalacak ve Görev 1 basitçe Engellenmiş durumundan Hazır (Ready) durumuna geçecektir. Bu senaryo, dikey çizgilerin bir tick kesmesinin (tick interrupt) meydana geldiği zamanları işaret ettiği Şekil 8.6'da gösterilmektedir.



Şekil 8.6 Aynı önceliğe sahip görevlerin aynı mutex'i kullanması durumunda olası bir yürütme dizisi

Şekil 8.6'da gösterilen senaryoda FreeRTOS zamanlayıcısı, mutex kullanılabilir (available) olur olmaz Görev 1'i Çalışıyor durumundaki görev yapmaz çünkü:

- Görev 1 ve Görev 2 aynı önceliğe sahiptir, bu nedenle Görev 2 Engellenmiş duruma girmediği sürece, bir sonraki tick kesmesine (tick interrupt) kadar (`FreeRTOSConfig.h`'de `configUSE_TIME_SLICING`'in 1 olarak ayarlandığı varsayılarak) Görev 1'e geçiş gerçekleşmemelidir.
- Bir görev, bir mutex'i dar bir döngüde (tight loop) kullanıyorsa ve görev mutex'i her "verdiğinde" (gave) bir bağlam geçişi (context switch) meydana geliyorsa, o zaman görev yalnızca kısa bir süre için Çalışıyor durumunda kalır. İki veya daha fazla görev aynı mutex'i dar bir döngüde kullanıyorsa, görevler arasında hızla geçiş yapılarak işlem süresi boşa harcanır (wasted).

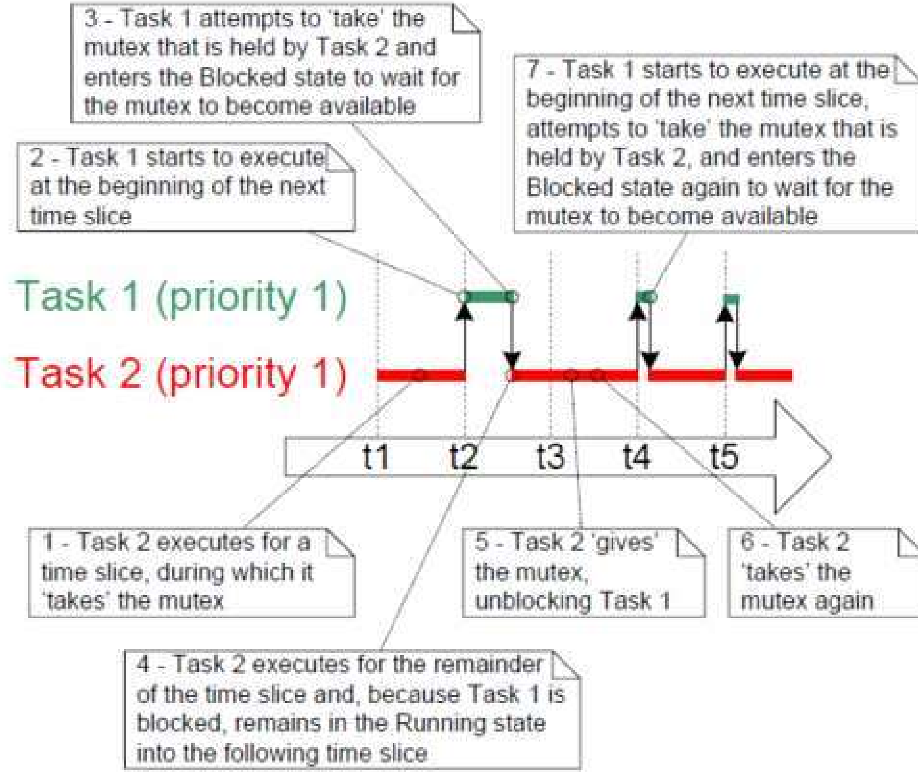
Bir mutex birden fazla görev tarafından dar bir döngüde kullanılıyorsa ve mutex'i kullanan görevler aynı önceliğe sahipse, görevlerin yaklaşık olarak eşit miktarda işlem süresi (processing time) almasını sağlamaya özen gösterilmelidir. Görevlerin eşit miktarda işlem süresi alamamasının nedeni, aynı öncelikte oluşturulmuş Liste 8.15'te gösterilen görevin iki örneği (instance) varsa ortaya çıkabilecek bir yürütme dizisini gösteren Şekil 8.7'de açıklanmıştır.

```
/* Dar bir döngüde (tight loop) mutex kullanan bir görevin uygulaması.

   Görev yerel bir arabellekte (local buffer) bir metin dizesi oluşturur,

   ardından dizeyi bir ekrana (display) yazar. Ekrana erişim bir mutex

   ile korunur. */

void vATask( void *pvParameter )

{

    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];


    for( ;; )

    {

        /* Metin dizesini üretin – bu hızlı bir işlemdir. */

        vGenerateTextInALocalBuffer( cTextBuffer );


        /* Ekrana erişimi koruyan mutex'i elde edin. */

        xSemaphoreTake( xMutex, portMAX_DELAY );


        /* Üretilen metni ekrana yazın – bu yavaş bir işlemdir. */

        vCopyTextToFrameBuffer( cTextBuffer );


        /* Metin ekrana yazılmıştır, bu nedenle mutex'i iade edin. */

        xSemaphoreGive( xMutex );

    }

}
```

Liste 8.15 Dar bir döngüde bir mutex kullanan görev

Liste 8.15'teki yorumlar, dize oluşturmanın hızlı bir işlem olduğunu ve ekranı güncellemenin yavaş bir işlem olduğunu belirtmektedir. Bu nedenle, ekran güncellenirken mutex elde tutulduğu (held) için, görev çalışma süresinin (run time) büyük bir kısmı boyunca mutex'i elinde tutacaktır.

Şekil 8.7'de dikey çizgiler tick kesmesinin (tick interrupt) meydana geldiği zamanları işaretler.



Şekil 8.7 Liste 8.15'te gösterilen görevin iki örneğinin aynı öncelikte oluşturulması halinde oluşabilecek yürütme dizisi

Şekil 8.7'deki 7. Adım, Görev 1'in tekrar Engellenmiş (Blocked) durumuna girmesini gösterir; bu, `xSemaphoreTake()` API işlevinin içinde gerçekleşir.

Şekil 8.7, bir zaman diliminin (time slice) başlangıcı, Görev 2'nin mutex sahibi olmadığı kısa dönemlerden birine denk gelene kadar Görev 1'in mutex'i elde etmesinin engelleneceğini göstermektedir.

Şekil 8.7'de gösterilen senaryo, `xSemaphoreGive()` çağrısından sonra `taskYIELD()` çağrısı eklenerek önlenebilir. Bu durum, görev mutex'i elinde tutarken tick sayımı (tick count) değiştiyse `taskYIELD()` çağrılan Liste 8.16'da gösterilmiştir.

```
void vFunction( void *pvParameter )

{

    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];
```

```
    TickType_t xTimeAtWhichMutexWasTaken;


    for( ;; )

    {

        /* Metin dizesini üretin – bu hızlı bir işlemdir. */

        vGenerateTextInALocalBuffer( cTextBuffer );


        /* Ekrana erişimi koruyan mutex'i elde edin. */

        xSemaphoreTake( xMutex, portMAX_DELAY );


        /* Mutex'in alındığı (taken) zamanı kaydedin. */

        xTimeAtWhichMutexWasTaken = xTaskGetTickCount();


        /* Üretilen metni ekrana yazın – bu yavaş bir işlemdir. */

        vCopyTextToFrameBuffer( cTextBuffer );


        /* Metin ekrana yazılmıştır, bu nedenle mutex'i iade edin. */

        xSemaphoreGive( xMutex );


        /* taskYIELD() işlevi her yinelemede (iteration) çağrılsaydı,

           bu görev Çalışıyor durumunda yalnızca kısa bir süre kalır

           ve görevler arasında hızla geçiş yapılarak işlem süresi boşa

           harcanırdı. Bu nedenle, taskYIELD() işlevini yalnızca mutex

           elde tutulurken (held) tick sayımı değiştiyse çağırın. */

        if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )

        {

            taskYIELD();

        }

    }
```

```
}
```

Liste 8.16 Döngüde (loop) bir mutex kullanan görevlerin daha eşit miktarda işlem süresi almasını sağlarken, aynı zamanda görevler arasında çok hızlı geçiş yapılarak işlem süresinin boşa harcanmamasını sağlamak

8.4 Kapı Tutucu Görevler (Gatekeeper Tasks)

Kapı tutucu görevler, öncelik ters çevirme (priority inversion) veya kilitlenme (deadlock) riski olmaksızın karşılıklı dışlama (mutual exclusion) sağlamak için temiz bir yöntem sunar.

Kapı tutucu görev (gatekeeper task), bir kaynağın tek sahibi olan bir görevdir. Kaynağa doğrudan erişmesine yalnızca kapı tutucu görevin izin verilir — kaynağa erişmesi gereken diğer herhangi bir görev, bunu yalnızca dolaylı olarak kapı tutucunun (gatekeeper) hizmetlerini (services) kullanarak yapabilir.

8.4.1 vPrintString() İşlevini Bir Kapı Tutucu Görev Kullanacak Şekilde Yeniden Yazma

Örnek 8.2, `vPrintString()` için başka bir alternatif uygulama (implementation) sağlar. Bu kez, standart çıktıya (standard out) erişimi yönetmek için bir kapı tutucu görev kullanılır. Bir görev standart çıktıya bir mesaj yazmak istediğinde, bir yazdırma işlevini (print function) doğrudan çağırmaz, bunun yerine mesajı kapı tutucuya (gatekeeper) gönderir.

Kapı tutucu görev, standart çıktıya erişimi serileştirmek (serialize) için bir FreeRTOS kuyruğu kullanır. Görevin dahili uygulamasının karşılıklı dışlamayı dikkate alması gerekmez, çünkü standart çıktıya doğrudan erişmesine izin verilen tek görev budur.

Kapı tutucu görev, zamanının çoğunu kuyrukta (queue) mesajların gelmesini bekleyerek Engellenmiş (Blocked) durumunda geçirir. Bir mesaj geldiğinde, kapı tutucu bir sonraki mesajı beklemek üzere Engellenmiş duruma dönmeden önce mesajı standart çıktıya yazar. Kapı tutucu görevin uygulanması Liste 8.18'de gösterilmiştir.

Kesmeler (interrupts) kuyruklara (queues) gönderme yapabilir, bu nedenle kesme servis rutinleri (interrupt service routines) terminale mesaj yazmak için kapı tutucu hizmetlerini güvenli bir şekilde kullanabilir. Bu örnekte, her 200 tick'te bir mesaj yazdırmak için bir tick kancası (tick hook) işlevi kullanılmıştır.

Bir tick kancası (veya tick geri çağırması - tick callback), her tick kesmesi sırasında çekirdek (kernel) tarafından çağrılan bir işlevdir. Bir tick kancası işlevini kullanmak için:

1. `FreeRTOSConfig.h` içinde `configUSE_TICK_HOOK` değerini 1 olarak ayarlayın.
2. Liste 8.17'de gösterilen tam işlev adını ve prototipini kullanarak kanca (hook) işlevinin uygulamasını (implementation) sağlayın.

```
void vApplicationTickHook( void );
```

Liste 8.17 Bir tick kancası (tick hook) işlevi için isim ve prototip

Tick kancası işlevleri tick kesmesi (tick interrupt) bağlamında (context) yürütülür ve bu nedenle çok kısa tutulmalı, yalnızca orta düzeyde bir yığın alanı (stack space) kullanmalı ve `'FromISR()'` ile bitmeyen hiçbir FreeRTOS API işlevini çağırmamalıdır.

Çizgeleyici (scheduler) her zaman tick kancası işlevinden hemen sonra yürütülecektir, bu nedenle tick kancasından çağrılan kesme güvenli (interrupt safe) FreeRTOS API işlevlerinin `pxHigherPriorityTaskWoken` parametrelerini kullanmaları gerekmez ve bu parametre `NULL` olarak ayarlanabilir.

```
static void prvStdioGatekeeperTask( void *pvParameters )

{

    char *pcMessageToPrint;



    /* Bu, standart çıktıya (standard out) yazmasına izin verilen

       tek görevdir. Çıktıya bir dize yazmak isteyen başka herhangi

       bir görev doğrudan standart çıktıya erişmez, bunun yerine

       dizeyi bu göreve gönderir. Standart çıktıya yalnızca bu görev

       eriştiğinden, görevin kendi uygulamasında dikkate alınması gereken

       karşılıklı dışlama (mutual exclusion) veya serileştirme (serialization)

       sorunları yoktur. */

    for( ;; )

    {

        /* Bir mesajın gelmesini bekleyin. Belirsiz (indefinite) bir

           blok süresi (block time) belirtilmiştir, bu nedenle dönüş

           değerini kontrol etmeye gerek yoktur – işlev yalnızca bir mesaj

           başarıyla alındığında dönecektir. */

        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );



        /* Alınan dizeyi çıktı olarak verin. */

        printf( "%s", pcMessageToPrint );
```

```
        fflush( stdout );


        /* Bir sonraki mesajı beklemek üzere döngüye geri dönün. */
    }
}
```

Liste 8.18 Kapı tutucu görev (gatekeeper task)

Örnek 8.2 Yazdırma görevi için alternatif uygulama

Kuyruğa yazan görev Liste 8.19'da gösterilmektedir. Daha önce olduğu gibi, görevin iki ayrı örneği (instance) oluşturulur ve görevin kuyruğa yazdığı dize (string), görev parametresi kullanılarak göreve geçirilir.

```
static void prvPrintTask( void *pvParameters )
{
    int iIndexToString;
    const TickType_t xMaxBlockTimeTicks = 0x20;


    /* Bu görevin iki örneği oluşturulur. Görev parametresi,
       bir dizeler dizisine (array of strings) indeks (index)
       geçirmek için kullanılır. Bunu gerekli türe (type) dönüştürün. */
    iIndexToString = ( int ) pvParameters;


    for( ;; )
    {
        /* Dizeyi doğrudan değil, bir kuyruk aracılığıyla kapı tutucu
           göreve dizenin bir işaretçisini (pointer) geçirerek yazdırın.
           Kuyruk, çizgeleyici başlatılmadan önce oluşturulur, böylece
           bu görev ilk kez yürütüldüğünde zaten var olacaktır. Bir blok
           süresi belirtilmemiştir çünkü kuyrukta her zaman yer olmalıdır. */
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ]), 0
);
```

```
        /* Sözde rastgele (pseudo random) bir süre bekleyin. rand() işlevinin
           ilgili olarak yeniden girilebilir (reentrant) olmadığını unutmayın,
           ancak bu durumda kodun hangi değerin döndürüldüğünü önemsememesi
           nedeniyle bunun pek bir önemi yoktur. Daha güvenli bir uygulamada,
           rand() işlevinin yeniden girilebilir olduğu bilinen bir sürümü
kullanılmalı
           veya rand() çağrıları kritik bir bölüm (critical section) kullanılarak
           korunmalıdır. */
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

Liste 8.19 Örnek 8.2 için yazdırma görevi uygulaması

Tick kancası işlevi (tick hook function) kaç kez çağrıldığını sayar ve sayım 200'e ulaştığında mesajını kapı tutucu (gatekeeper) göreve gönderir. Yalnızca gösterim (demonstration) amacıyla, tick kancası kuyruğun önüne (front of the queue), görevler ise kuyruğun arkasına (back of the queue) yazar. Tick kancasının uygulaması Liste 8.20'de gösterilmiştir.

```
void vApplicationTickHook( void )
{
    static int iCount = 0;

    /* Her 200 tick'te bir mesaj yazdırın. Mesaj doğrudan yazdırılmaz,
       kapı tutucu göreve gönderilir. */
    iCount++;
    if( iCount >= 200 )
    {
        /* xQueueSendToFrontFromISR() tick kancasından çağrıldığı için
           xHigherPriorityTaskWoken parametresini (üçüncü parametre)
           kullanmaya gerek yoktur ve parametre NULL olarak ayarlanmıştır. */
```

```
        xQueueSendToFrontFromISR( xPrintQueue,
                                  &( pcStringsToPrint[ 2 ] ),
                                  NULL );

        /* Dizeyi 200 tick sonra tekrar yazdırmaya hazır olmak için sayımı
           sıfırlayın. */
        iCount = 0;
    }
}
```

Liste 8.20 Tick kancası (tick hook) uygulaması

Normalde olduğu gibi, `main()` örneği çalıştırmak için gerekli kuyrukları ve görevleri oluşturur, ardından çizgeleyiciyi başlatır. `main()` işlevinin uygulaması Liste 8.21'de gösterilmiştir.

```
/* Görevlerin ve kesmenin kapı tutucu aracılığıyla yazdıracağı
   dizeleri tanımlayın. */
static char *pcStringsToPrint[] =
{
    "Task 1 ****************************************************\r\n",
    "Task 2 ----------------------------------------------------\r\n",
    "Message printed from the tick hook interrupt ##############\r\n"
};

/*-----------------------------------------------------------*/

/* QueueHandle_t türünde bir değişken bildirin (declare). Kuyruk,
   yazdırma görevlerinden ve tick kesmesinden kapı tutucu göreve
   mesaj göndermek için kullanılır. */
QueueHandle_t xPrintQueue;
```

```
/*-----------------------------------------------------------*/

int main( void )
{
    /* Bir kuyruk kullanılmadan önce açıkça oluşturulmalıdır. Kuyruk,
       maksimum 5 karakter işaretçisi (character pointer) tutacak şekilde
       oluşturulur. */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* Kuyruğun başarıyla oluşturulduğunu kontrol edin. */
    if( xPrintQueue != NULL )
    {
        /* Kapı tutucuya mesaj gönderen görevlerin iki örneğini (instance)
oluşturun.
           Görevin kullandığı dizeye giden indeks, görev parametresi
           (xTaskCreate() işlevine giden 4. parametre) aracılığıyla göreve
geçirilir.
           Görevler farklı önceliklerde oluşturulur, böylece yüksek öncelikli
görev
           ara sıra düşük öncelikli görevi engeller (preempt). */
        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Kapı tutucu görevini oluşturun. Doğrudan standart çıktıya (standard
out)
           erişmesine izin verilen tek görev budur. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL );

        /* Oluşturulan görevlerin çalışmaya başlaması için çizgeleyiciyi başlatın.
*/
        vTaskStartScheduler();
    }
```

```
    /* Her şey yolundaysa çizgeleyici artık görevleri çalıştıracağından
       main() buraya asla ulaşmayacaktır. main() buraya ulaşırsa,
       boşta kalan görevin (idle task) oluşturulması için yeterli
       yığın (heap) belleği bulunmamış olması muhtemeldir. Bölüm 3
       yığın belleği yönetimi hakkında daha fazla bilgi sağlar. */
    for( ;; );
}
```

Liste 8.21 Örnek 8.2 için `main()` işlevinin uygulanması

Örnek 8.2 çalıştırıldığında üretilen çıktı Şekil 8.8'de gösterilmektedir. Görülebileceği gibi, görevlerden kaynaklanan dizeler ve kesmeden kaynaklanan dizelerin tümü hiçbir bozulma (corruption) olmadan doğru şekilde yazdırılmaktadır.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Message printed from the tick hook interrupt ##############
Message printed from the tick hook interrupt ##############
Task 2 -----------------------------------------------
Task 1 ***********************************************
Message printed from the tick hook interrupt ##############
Task 1 ***********************************************
Message printed from the tick hook interrupt ##############
Task 2 -----------------------------------------------
Message printed from the tick hook interrupt ##############
Task 2 -----------------------------------------------
Task 1 ***********************************************
Task 1 ***********************************************
Message printed from the tick hook interrupt ##############
Message printed from the tick hook interrupt ##############
Task 2 -----------------------------------------------
Task 1 ***********************************************
Message printed from the tick hook interrupt ##############
Task 2 -----------------------------------------------
Task 2 -----------------------------------------------
Task 2 -----------------------------------------------
Task 1 ***********************************************
Task 2 -----------------------------------------------
Message printed from the tick hook interrupt ##############
Task 1 ***********************************************
```

Şekil 8.8 Örnek 8.2 yürütüldüğünde üretilen çıktı

Kapı tutucu (gatekeeper) göreve yazdırma görevlerinden daha düşük bir öncelik atanmıştır — bu nedenle kapı tutucuya gönderilen mesajlar, her iki yazdırma görevi de Engellenmiş durumuna (Blocked state) geçene kadar kuyrukta kalır. Bazı durumlarda, kapı tutucuya daha yüksek bir öncelik atamak uygun olabilir, böylece mesajlar hemen işlenir — ancak bunu yapmak, korunan kaynağa erişimi tamamlanana kadar kapı tutucunun düşük öncelikli görevleri geciktirmesi pahasına (cost) olacaktır.

9 Olay Grupları (Event Groups)

9.1 Bölüm Girişi ve Kapsamı

Gerçek zamanlı gömülü sistemlerin (real-time embedded systems) olaylara (events) yanıt olarak eyleme geçmesi gerektiği daha önce belirtilmişti. Önceki bölümlerde, olayların görevlere (tasks) iletilmesine olanak tanıyan FreeRTOS özellikleri açıklanmıştır. Bu tür özelliklere örnek olarak, her ikisi de aşağıdaki özelliklere sahip olan semaforlar (semaphores) ve kuyruklar (queues) verilebilir:

- Bir görevin tek bir olayın gerçekleşmesini beklemek üzere Engellenmiş durumunda (Blocked state) kalmasına olanak tanırlar.
- Olay gerçekleştiğinde tek bir görevi (single task) Engellenmiş durumundan çıkarırlar (unblock). Engellenmiş durumundan çıkarılan görev, olayı bekleyen en yüksek öncelikli görevdir.

Olay grupları (Event groups), olayların görevlere iletilmesine olanak tanıyan bir başka FreeRTOS özelliğidir. Kuyrukların ve semaforların aksine:

- Olay grupları, bir görevin bir veya birden fazla olayın bir kombinasyonunun gerçekleşmesini beklemek üzere Engellenmiş durumunda kalmasına olanak tanır.
- Olay grupları, olay gerçekleştiğinde aynı olayı veya olaylar kombinasyonunu bekleyen tüm görevleri Engellenmiş durumundan çıkarır.

Olay gruplarının bu benzersiz özellikleri; birden fazla görevi senkronize etmek (synchronize), olayları birden fazla göreve yayınlamak (broadcasting), bir görevin bir dizi olaydan herhangi birinin gerçekleşmesini beklemek üzere Engellenmiş durumunda kalmasına olanak sağlamak ve bir görevin birden fazla işlemin tamamlanmasını beklemek üzere Engellenmiş durumunda kalmasına olanak sağlamak için onları yararlı kılar.

Olay grupları ayrıca, çoğu zaman birçok ikili semaforu (binary semaphore) tek bir olay grubuyla değiştirmek mümkün olduğundan, bir uygulamanın kullandığı RAM'i azaltma fırsatı sunar.

Olay grubu işlevselliği isteğe bağlıdır (optional). Olay grubu işlevselliğini dahil etmek için, FreeRTOS kaynak dosyası `event_groups.c`'yi projenizin bir parçası olarak derleyin.

9.1.1 Kapsam

Bu bölüm okuyuculara aşağıdakiler hakkında iyi bir anlayış kazandırmayı amaçlamaktadır:

- Olay gruplarının pratik kullanım alanları.
- Diğer FreeRTOS özelliklerine kıyasla olay gruplarının avantajları ve dezavantajları.

- Bir olay grubundaki bitlerin (bits) nasıl ayarlanacağı (set).
- Bir olay grubunda bitlerin ayarlanmasını beklemek için Engellenmiş durumunda nasıl bekleneceği.
- Bir dizi görevi senkronize etmek için bir olay grubunun nasıl kullanılacağı.

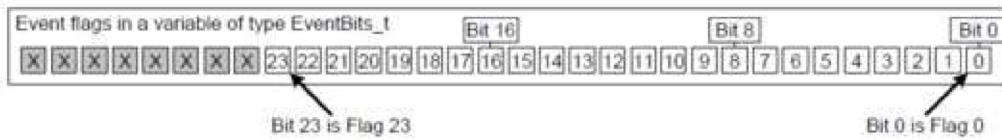
9.2 Bir Olay Grubunun Özellikleri

9.2.1 Olay Grupları, Olay Bayrakları ve Olay Bitleri

Bir olay 'bayrağı' (event flag), bir olayın meydana gelip gelmediğini göstermek için kullanılan Boolean (1 veya 0) bir değerdir. Bir olay 'grubu' (event group) bir olay bayrakları kümesidir (set of event flags).

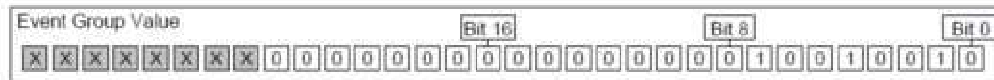
Bir olay bayrağı yalnızca 1 veya 0 olabilir, bu da bir olay bayrağının durumunun tek bir bit'te ve bir olay grubundaki tüm olay bayraklarının durumunun tek bir değişkende saklanmasına olanak tanır; bir olay grubundaki her bir olay bayrağının durumu, `EventBits_t` türünde bir değişkende tek bir bitle temsil edilir. Bu nedenle, olay bayrakları aynı zamanda olay 'bitleri' (event bits) olarak da bilinir. `EventBits_t` değişkeninde bir bit 1 olarak ayarlanmışsa (set), o bitin temsil ettiği olay meydana gelmiştir. `EventBits_t` değişkeninde bir bit 0 olarak ayarlanmışsa (temizlenmişse - clear), o bitin temsil ettiği olay meydana gelmemiştir.

Şekil 9.1, `EventBits_t` türündeki bir değişkende ayrı olay bayraklarının ayrı bitlere nasıl eşlendiğini (mapped) göstermektedir.



Şekil 9.1 `EventBits_t` türünde bir değişkende olay bayrağının bit numarası ile eşlenmesi

Örnek olarak, bir olay grubunun değeri `0x92` (ikili 1001 0010) ise, o zaman yalnızca 1, 4 ve 7 numaralı olay bitleri ayarlanmıştır, yani yalnızca 1, 4 ve 7 numaralı bitler tarafından temsil edilen olaylar gerçekleşmiştir. Şekil 9.2, 1, 4 ve 7 numaralı olay bitlerinin ayarlanmış olduğu ve diğer tüm olay bitlerinin temizlendiği (clear) ve böylece olay grubuna `0x92` değerinin verildiği `EventBits_t` türünde bir değişkeni göstermektedir.



Şekil 9.2 Yalnızca 1, 4 ve 7 numaralı bitlerin ayarlandığı ve diğer tüm olay bayraklarının temizlendiği, olay grubunun değerini `0x92` yapan bir olay grubu

Bir olay grubu içindeki tek tek bitlere bir anlam atamak uygulama yazarının (application writer) sorumluluğundadır. Örneğin, uygulama yazarı bir olay grubu oluşturabilir ve ardından:

- Olay grubundaki bit 0'ı "ağdan (network) bir mesaj alındı" anlamına gelecek şekilde tanımlayabilir.
- Olay grubundaki bit 1'i "bir mesaj ağa gönderilmeye hazır" anlamına gelecek şekilde tanımlayabilir.
- Olay grubundaki bit 2'yi "mevcut ağ bağlantısını iptal et (abort)" anlamına gelecek şekilde tanımlayabilir.

9.2.2 EventBits_t Veri Türü Hakkında Daha Fazla Bilgi

Bir olay grubundaki olay bitlerinin sayısı, `FreeRTOSConfig.h` içindeki `configTICK_TYPE_WIDTH_IN_BITS` derleme zamanı (compile time) yapılandırma sabitine bağlıdır:

- `configTICK_TYPE_WIDTH_IN_BITS`, `TICK_TYPE_WIDTH_16_BITS` ise, her bir olay grubu 8 kullanılabilir olay biti içerir.
- `configTICK_TYPE_WIDTH_IN_BITS`, `TICK_TYPE_WIDTH_32_BITS` ise, her bir olay grubu 24 kullanılabilir olay biti içerir.
- `configTICK_TYPE_WIDTH_IN_BITS`, `TICK_TYPE_WIDTH_64_BITS` ise, her bir olay grubu 56 kullanılabilir olay biti içerir.

(Not: `configTICK_TYPE_WIDTH_IN_BITS` normalde RTOS tick sayımını tutmak için kullanılan türü yapılandırır. Olay grupları özelliğiyle ilgisiz gibi görünse de, `EventBits_t` türü üzerindeki etkisi FreeRTOS'un dahili uygulamasının bir sonucudur. `configTICK_TYPE_WIDTH_IN_BITS` sabitinin `TICK_TYPE_WIDTH_16_BITS` olarak ayarlanması, yalnızca FreeRTOS 16 bitlik türleri 32 bitlik türlerden daha verimli bir şekilde işleyebilen bir mimaride yürütülürken yapılmalıdır.)

9.2.3 Birden Fazla Görev Tarafından Erişim (Access by Multiple Tasks)

Olay grupları, varlıklarından (existence) haberdar olan herhangi bir görev veya ISR tarafından erişilebilen kendi başlarına nesnelerdir (objects). Herhangi bir sayıda görev aynı olay grubunda bitler ayarlayabilir ve herhangi bir sayıda görev aynı olay grubundan bitler okuyabilir.

9.2.4 Bir Olay Grubunun Kullanımına İlişkin Pratik Bir Örnek

FreeRTOS+TCP TCP/IP yığınının (stack) uygulaması, bir olay grubunun aynı anda bir tasarımı basitleştirmek ve kaynak kullanımını (resource usage) en aza indirmek için nasıl kullanılabileceğine dair pratik bir örnek sunar.

Bir TCP soketi (socket) birçok farklı olaya yanıt vermelidir. Olay örnekleri arasında accept (kabul et), bind (bağla), read (oku) ve close (kapat) olayları bulunur. Bir soketin

herhangi bir zamanda bekleyebileceği olaylar, soketin durumuna bağlıdır. Örneğin, bir soket oluşturulmuşsa, ancak henüz bir adrese bağlanmamışsa (bound), bir bağlama olayı (bind event) almayı bekleyebilir, ancak bir okuma olayı (read event) almayı beklemez (bir adresi yoksa veri okuyamaz).

Bir FreeRTOS+TCP soketinin durumu `FreeRTOS_Socket_t` adı verilen bir yapıda (structure) tutulur. Yapı, soketin işlemesi (process) gereken her bir olay için tanımlanmış bir olay bitine sahip bir olay grubu içerir. Bir olayı veya olaylar grubunu beklemek için engellenen (block) FreeRTOS+TCP API çağrıları, basitçe olay grubunda engellenirler.

Olay grubu ayrıca bir 'iptal' (abort) biti içerir, bu da o sırada soketin hangi olayı beklediğine bakılmaksızın bir TCP bağlantısının iptal edilmesini sağlar.

9.3 Olay Gruplarını Kullanarak Olay Yönetimi

9.3.1 xEventGroupCreate() API İşlevi

FreeRTOS ayrıca derleme zamanında (compile time) statik olarak bir olay grubu oluşturmak için gereken belleği tahsis eden `xEventGroupCreateStatic()` işlevini de içerir: Bir olay grubu kullanılmadan önce açıkça oluşturulmalıdır.

Olay gruplarına `EventGroupHandle_t` türündeki değişkenler kullanılarak başvurulur (referenced). `xEventGroupCreate()` API işlevi bir olay grubu oluşturmak için kullanılır ve oluşturduğu olay grubuna başvurmak için bir `EventGroupHandle_t` döndürür.

```
EventGroupHandle_t xEventGroupCreate( void );
```

Liste 9.1 `xEventGroupCreate()` API işlevi prototipi

`xEventGroupCreate()` dönüş değeri:

- **Dönüş Değeri:** Eğer `NULL` döndürülürse, FreeRTOS'un olay grubu veri yapılarını tahsis etmesi (allocate) için mevcut yığın (heap) belleği yetersiz olduğundan olay grubu oluşturulamaz. (Bölüm 3 yığın belleği yönetimi hakkında daha fazla bilgi sağlamaktadır.)

9.3.2 xEventGroupSetBits() API İşlevi

`xEventGroupSetBits()` API işlevi, bir olay grubunda bir veya daha fazla bit ayarlar (sets) ve tipik olarak ayarlanmakta olan bit veya bitler tarafından temsil edilen olayların gerçekleştiğini bir göreve bildirmek (notify) için kullanılır.

Not: `xEventGroupSetBits()` işlevini asla bir kesme servis rutininden (ISR) çağırmayın. Bunun yerine kesme güvenli sürümü olan `xEventGroupSetBitsFromISR()` kullanılmalıdır.

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,

                                 const EventBits_t uxBitsToSet );
```

Liste 9.2 `xEventGroupSetBits()` API işlevi prototipi

`xEventGroupSetBits()` parametreleri ve dönüş değeri:

- **xEventGroup:** İçinde bitlerin ayarlandığı olay grubunun tanıtıcısıdır (handle). Olay grubu tanıtıcısı, olay grubunu oluşturmak için kullanılan `xEventGroupCreate()` çağrısından döndürülmüş olacaktır.
- **`uxBitsToSet`:** Olay grubunda 1 olarak ayarlanacak (set) olay bitini veya olay bitlerini belirten bir bit maskesidir (bit mask). Olay grubunun değeri, olay grubunun mevcut değeri ile `uxBitsToSet` içinde aktarılan değerin bitsel VEYA (bitwise OR) işlemine tabi tutulmasıyla güncellenir.
  Örnek olarak, `uxBitsToSet` değerini `0x04` (ikili 0100) olarak ayarlamak, olay grubundaki diğer tüm olay bitlerini değiştirmeden bırakırken, olay grubundaki olay biti 3'ün ayarlanmasına (zaten ayarlanmamışsa) neden olacaktır.
- **Dönüş Değeri:** `xEventGroupSetBits()` çağrısının döndüğü zamandaki olay grubunun değeri. Döndürülen değerin, `uxBitsToSet` tarafından belirtilen bitlerin ayarlanmış (set) olmasını zorunlu olarak içermeyebileceğini unutmayın, çünkü bitler farklı bir görev tarafından tekrar temizlenmiş (cleared) olabilir.

9.3.3 xEventGroupSetBitsFromISR() API İşlevi

`xEventGroupSetBitsFromISR()`, `xEventGroupSetBits()` işlevinin kesme güvenli (interrupt safe) sürümüdür.

Bir semafor vermek (give a semaphore) deterministik (deterministic) bir işlemdir çünkü semafor vermenin en fazla bir görevin Engellenmiş durumundan çıkmasına neden olabileceği önceden bilinmektedir. Bir olay grubunda bitler ayarlandığında, kaç görevin Engellenmiş durumundan çıkacağı önceden bilinmez, bu nedenle bir olay grubunda bitleri ayarlamak deterministik bir işlem değildir.

FreeRTOS tasarım ve uygulama standardı, bir kesme servis rutini içinde veya kesmeler devre dışı bırakıldığında deterministik olmayan işlemlerin (non-deterministic operations) gerçekleştirilmesine izin vermez. Bu nedenle, `xEventGroupSetBitsFromISR()` olay bitlerini doğrudan kesme servis rutini içinde ayarlamaz, bunun yerine eylemi RTOS arka plan (daemon) görevine devreder (defers).

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,

                                       const EventBits_t uxBitsToSet,
```

```
                                          BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 9.3 `xEventGroupSetBitsFromISR()` API işlevi prototipi

`xEventGroupSetBitsFromISR()` parametreleri ve dönüş değeri:

- **xEventGroup:** Bitlerin ayarlandığı olay grubunun tanıtıcısıdır.
- **`uxBitsToSet`:** Olay grubunda 1 olarak ayarlanacak olay bitini veya bitlerini belirten bit maskesidir. Örnek olarak, `uxBitsToSet` değerini `0x05` (ikili 0101) olarak ayarlamak, diğer bitleri değiştirmeden olay grubundaki olay biti 2 ve olay biti 0'ın ayarlanmasına neden olur.
- **`pxHigherPriorityTaskWoken`:** `xEventGroupSetBitsFromISR()` olay bitlerini doğrudan kesme servis rutini içinde ayarlamaz, bunun yerine zamanlayıcı komut kuyruğuna (timer command queue) bir komut göndererek eylemi RTOS arka plan (daemon) görevine erteler. Eğer arka plan görevi zamanlayıcı komut kuyruğunda verinin hazır olmasını beklemek üzere Engellenmiş (Blocked) durumundaysa, zamanlayıcı komut kuyruğuna yazmak arka plan görevinin Engellenmiş durumundan çıkmasına neden olacaktır. Eğer arka plan görevinin önceliği, o anda yürütülmekte olan görevin (kesintiye uğratılan görevin) önceliğinden yüksekse, o zaman `xEventGroupSetBitsFromISR()` dahili olarak `*pxHigherPriorityTaskWoken` değişkenini `pdTRUE` olarak ayarlayacaktır.
  Eğer `xEventGroupSetBitsFromISR()` bu değeri `pdTRUE` olarak ayarlarsa, kesmeden çıkılmadan önce bir bağlam geçişi (context switch) gerçekleştirilmelidir. Bu, arka plan görevi Hazır (Ready) durumundaki en yüksek öncelikli görev olacağından, kesmenin doğrudan arka plan görevine dönmesini sağlayacaktır.
- **Dönüş Değeri:** İki olası dönüş değeri vardır:
  - Veri, zamanlayıcı komut kuyruğuna başarıyla gönderildiyse yalnızca `pdPASS` döndürülür.
  - Kuyruk zaten dolu olduğu için 'bitleri ayarla' (set bits) komutu zamanlayıcı komut kuyruğuna yazılamadıysa `pdFALSE` döndürülür.

9.3.4 xEventGroupWaitBits() API İşlevi

`xEventGroupWaitBits()` API işlevi, bir görevin bir olay grubunun değerini okumasına ve isteğe bağlı olarak olay bitleri zaten ayarlanmamışsa, olay grubundaki bir veya daha fazla olay bitinin ayarlanmasını beklemek üzere Engellenmiş (Blocked) durumunda kalmasına olanak tanır.

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup,

                                 const EventBits_t uxBitsToWaitFor,

                                 const BaseType_t xClearOnExit,

                                 const BaseType_t xWaitForAllBits,

                                 TickType_t xTicksToWait );
```

Liste 9.4 `xEventGroupWaitBits()` API işlevi prototipi

Çizgeleyici (scheduler) tarafından bir görevin Engellenmiş durumuna girip girmeyeceğini ve bir görevin Engellenmiş durumundan ne zaman çıkacağını belirlemek için kullanılan koşula 'bloktan çıkarma koşulu' (unblock condition) denir. Bloktan çıkarma koşulu, `uxBitsToWaitFor` ve `xWaitForAllBits` parametre değerlerinin bir kombinasyonu ile belirlenir:

- `uxBitsToWaitFor`, olay grubunda hangi olay bitlerinin test edileceğini belirtir.
- `xWaitForAllBits`, bitsel VEYA (bitwise OR) testinin mi yoksa bitsel VE (bitwise AND) testinin mi kullanılacağını belirtir.

Bir görev, `xEventGroupWaitBits()` çağrıldığında bloktan çıkarma koşulu sağlanmışsa (met) Engellenmiş duruma girmeyecektir.

Tablo 5'te, bir görevin Engellenmiş durumuna girmesine veya Engellenmiş durumundan çıkmasına neden olacak koşullara örnekler verilmiştir. Tablo 5 sadece olay grubunun ve `uxBitsToWaitFor` değerlerinin en anlamsız (least significant) dört ikili bitini gösterir - bu iki değerin diğer bitlerinin sıfır olduğu varsayılır.

Mevcut Olay Grubu Değeri	**`uxBitsToWaitFor` değeri**	**`xWaitForAllBits` değeri**	**Sonuç Olarak Ortaya Çıkan Davranış (Resultant Behavior)**
0000	0101	`pdFALSE`	Olay grubunda bit 0 veya bit 2 ayarlanmadığı için çağıran görev (calling task) Engellenmiş durumuna girecek ve olay grubunda bit 0 VEYA bit 2 ayarlandığında Engellenmiş durumundan çıkacaktır.
0100	0101	`pdTRUE`	Olay grubunda bit 0 ve bit 2'nin her ikisi de ayarlanmadığı için çağıran görev Engellenmiş durumuna girecek ve olay grubunda hem bit 0 VE hem de bit 2 ayarlandığında Engellenmiş durumundan çıkacaktır.
0100	0110	`pdFALSE`	`xWaitForAllBits` `pdFALSE` olduğundan ve `uxBitsToWaitFor` tarafından

			belirtilen iki bitten biri olay grubunda zaten ayarlanmış olduğundan çağıran görev Engellenmiş durumuna girmeyecektir.
0100	0110	pdTRUE	xWaitForAllBits pdTRUE olduğundan ve uxBitsToWaitFor tarafından belirtilen iki bitten yalnızca biri olay grubunda önceden ayarlanmış olduğundan çağıran görev Engellenmiş durumuna girecektir. Görev, olay grubunda hem bit 1 hem de bit 2 ayarlandığında Engellenmiş durumundan çıkacaktır.

Tablo 5 uxBitsToWaitFor ve xWaitForAllBits Parametrelerinin Etkisi

Çağıran görev (calling task), test edilecek bitleri uxBitsToWaitFor parametresini kullanarak belirler ve büyük olasılıkla çağıran görevin bloktan çıkarma koşulu (unblock condition) sağlandıktan sonra bu bitleri tekrar sıfıra temizlemesi (clear) gerekecektir. Olay bitleri xEventGroupClearBits() API işlevi kullanılarak temizlenebilir, ancak olay bitlerini manuel olarak temizlemek için bu işlevi kullanmak, aşağıdaki durumlarda uygulama kodunda yarış koşullarına (race conditions) yol açacaktır:

- Aynı olay grubunu kullanan birden fazla görev varsa.
- Bitler olay grubunda farklı bir görev tarafından veya bir kesme servis rutini tarafından ayarlanmışsa.

Bu potansiyel yarış koşullarından kaçınmak için xClearOnExit parametresi sağlanmıştır. xClearOnExit parametresi pdTRUE olarak ayarlanırsa, olay bitlerinin test edilmesi ve temizlenmesi çağıran görev için atomik (diğer görevler veya kesmeler tarafından kesintiye uğratılamayan) bir işlem gibi görünür.

xEventGroupWaitBits() parametreleri ve dönüş değeri:

- **xEventGroup:** Okunmakta olan olay bitlerini içeren olay grubunun tanıtıcısıdır (handle). Olay grubu tanıtıcısı, olay grubunu oluşturmak için kullanılan xEventGroupCreate() çağrısından döndürülmüş olacaktır.
- **uxBitsToWaitFor:** Olay grubunda test edilecek olay bitini veya olay bitlerini belirten bir bit maskesidir (bit mask).
  Örneğin, çağıran görev olay grubunda olay biti 0 ve/veya olay biti 2'nin ayarlanmasını beklemek istiyorsa, uxBitsToWaitFor değerini 0x05 (ikili 0101) olarak ayarlayın. Daha fazla örnek için Tablo 5'e bakın.
- **xClearOnExit:** Çağıran görevin bloktan çıkarma koşulu (unblock condition) sağlanmışsa ve xClearOnExit, pdTRUE olarak ayarlanmışsa, uxBitsToWaitFor

tarafından belirtilen olay bitleri, çağıran görev `xEventGroupWaitBits()` API işlevinden çıkmadan önce olay grubunda tekrar 0'a temizlenecektir (cleared). Eğer `xClearOnExit`, `pdFALSE` olarak ayarlanmışsa, olay grubundaki olay bitlerinin durumu `xEventGroupWaitBits()` API işlevi tarafından değiştirilmez.

- **`xWaitForAllBits`:** `uxBitsToWaitFor` parametresi, olay grubunda test edilecek olay bitlerini belirtir. `xWaitForAllBits`, çağıran görevin `uxBitsToWaitFor` parametresi tarafından belirtilen olay bitlerinden bir veya daha fazlası ayarlandığında mı, yoksa yalnızca `uxBitsToWaitFor` parametresi tarafından belirtilen olay bitlerinin **tümü** ayarlandığında mı Engellenmiş durumundan çıkarılması gerektiğini belirtir.
  Eğer `xWaitForAllBits`, `pdFALSE` olarak ayarlanırsa, bloktan çıkarma koşulunun sağlanmasını beklemek üzere Engellenmiş duruma giren bir görev, `uxBitsToWaitFor` tarafından belirtilen bitlerden herhangi biri ayarlandığında (veya `xTicksToWait` parametresi tarafından belirtilen zaman aşımı süresi dolduğunda) Engellenmiş durumundan çıkacaktır.
  Eğer `xWaitForAllBits`, `pdTRUE` olarak ayarlanırsa, bloktan çıkarma koşulunun sağlanmasını beklemek üzere Engellenmiş duruma giren bir görev, **yalnızca** `uxBitsToWaitFor` tarafından belirtilen bitlerin tümü ayarlandığında (veya `xTicksToWait` parametresi tarafından belirtilen zaman aşımı süresi dolduğunda) Engellenmiş durumundan çıkacaktır.
  Örnekler için Tablo 5'e bakın.
- **`xTicksToWait`:** Görevin bloktan çıkarma koşulunun sağlanmasını beklemek için Engellenmiş durumunda kalması gereken maksimum süredir.
  Eğer `xTicksToWait` sıfır ise veya `xEventGroupWaitBits()` çağrıldığı sırada bloktan çıkarma koşulu sağlanmışsa, `xEventGroupWaitBits()` hemen dönecektir (return immediately).
  Blok süresi (block time) tick periyotları (tick periods) cinsinden belirtilir, bu nedenle temsil ettiği mutlak zaman tick frekansına bağlıdır. `pdMS_TO_TICKS()` makrosu, milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için kullanılabilir.
  `FreeRTOSConfig.h` dosyasında `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak görevin süresiz olarak (zaman aşımı olmadan) beklemesine neden olacaktır.
- **Döndürülen Değer:**
  - `xEventGroupWaitBits()` işlevi çağıran görevin bloktan çıkarma koşulu sağlandığı (met) için döndüyse, döndürülen değer çağıran görevin bloktan çıkarma koşulunun sağlandığı andaki olay grubunun değeridir (`xClearOnExit`, `pdTRUE` ise herhangi bir bit otomatik olarak temizlenmeden önce). Bu durumda döndürülen değer aynı zamanda bloktan çıkarma koşulunu da sağlayacaktır.
  - `xEventGroupWaitBits()` işlevi, `xTicksToWait` parametresi tarafından belirtilen blok süresi dolduğu için döndüyse, döndürülen değer blok süresinin dolduğu andaki olay grubunun değeridir. Bu durumda döndürülen değer bloktan çıkarma koşulunu sağlamayacaktır.

9.3.5 xEventGroupGetStaticBuffer() API İşlevi

`xEventGroupGetStaticBuffer()` API işlevi, statik olarak (statically) oluşturulmuş bir olay grubunun arabelleğine (buffer) işaret eden bir işaretçi (pointer) almak için bir yöntem

sağlar. Bu, olay grubunun oluşturulması sırasında sağlanan arabelleğin aynısıdır. *Not: `xEventGroupGetStaticBuffer()` işlevini asla bir kesme servis rutininden (interrupt service routine) çağırmayın.

```
BaseType_t xEventGroupGetStaticBuffer( EventGroupHandle_t xEventGroup,

                                       StaticEventGroup_t ** ppxEventGroupBuffer
);
```

Liste 9.5 `xEventGroupGetStaticBuffer()` API işlevi prototipi

`xEventGroupGetStaticBuffer()` parametreleri ve dönüş değeri:

- **xEventGroup:** Arabelleği (buffer) alınacak olay grubudur. Bu olay grubu `xEventGroupCreateStatic()` tarafından oluşturulmuş olmalıdır.
- **ppxEventGroupBuffer:** Olay grubunun veri yapısı arabelleğine bir işaretçi (pointer) döndürmek için kullanılır. Oluşturma sırasında sağlanan arabelleğin aynısıdır.
- **Dönüş Değeri:** İki olası dönüş değeri vardır:
    - Arabellek başarıyla alındıysa `pdTRUE` döndürülür.
    - Arabellek başarıyla alınamadıysa `pdFALSE` döndürülür.

Örnek 9.1 Olay gruplarıyla deneme (Experimenting with event groups)

Bu örnek şunların nasıl yapılacağını gösterir:

- Bir olay grubu oluşturmak.
- Bir kesme servis rutininden bir olay grubundaki bitleri ayarlamak.
- Bir görevden bir olay grubundaki bitleri ayarlamak.
- Bir olay grubunda engellenmek (Block).

`xEventGroupWaitBits()` işlevinin `xWaitForAllBits` parametresinin etkisi, önce örneğin `xWaitForAllBits` parametresi `pdFALSE` olarak ayarlanmış şekilde yürütülmesiyle ve ardından örneğin `xWaitForAllBits` parametresi `pdTRUE` olarak ayarlanmış şekilde yürütülmesiyle gösterilmektedir.

Olay biti 0 ve olay biti 1 bir görevden (task) ayarlanır. Olay biti 2 bir kesme servis rutininden (ISR) ayarlanır. Bu üç bite, Liste 9.6'da gösterilen `#define` ifadeleri (statements) kullanılarak açıklayıcı adlar verilmiştir.

```
/* Olay grubundaki olay bitleri için tanımlar. */

#define mainFIRST_TASK_BIT  ( 1UL << 0UL ) /* Olay biti 0, bir görev tarafından
ayarlanır */

#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Olay biti 1, bir görev tarafından
ayarlanır */
```

```
#define mainISR_BIT         ( 1UL << 2UL ) /* Olay biti 2, bir ISR tarafından
ayarlanır */
```

Liste 9.6 Örnek 9.1'de kullanılan olay biti tanımları

Liste 9.7, olay biti 0'ı ve olay biti 1'i ayarlayan görevin uygulanmasını göstermektedir. Bu görev bir döngüde yer alır, her bir `xEventGroupSetBits()` çağrısı arasında 200 milisaniyelik bir gecikmeyle arka arkaya önce bir biti sonra diğer biti ayarlar. Yürütme dizisinin (sequence of execution) konsolda (console) görülebilmesini sağlamak için her bit ayarlanmadan önce bir dize (string) yazdırılır.

```
static void vEventBitSettingTask( void *pvParameters )
{
    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;

    for( ;; )
    {
        /* Bir sonraki döngüye başlamadan önce kısa bir süre gecikme yapın. */
        vTaskDelay( xDelay200ms );

        /* Görev tarafından olay biti 0'ın ayarlanmak üzere olduğunu bildiren
           bir mesaj yazdırın, ardından olay biti 0'ı ayarlayın. */
        vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );
        xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );

        /* Diğer biti ayarlamadan önce kısa bir süre gecikme yapın. */
        vTaskDelay( xDelay200ms );

        /* Görev tarafından olay biti 1'in ayarlanmak üzere olduğunu bildiren
           bir mesaj yazdırın, ardından olay biti 1'i ayarlayın. */
        vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );
        xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );
    }
```

```
}
```

Liste 9.7 Örnek 9.1'de olay grubundaki iki biti ayarlayan görev

Liste 9.8, olay grubundaki bit 2'yi ayarlayan kesme servis rutininin (ISR) uygulanmasını göstermektedir. Yine, yürütme dizisinin konsolda görülebilmesini sağlamak için bit ayarlanmadan önce bir dize yazdırılır. Ancak bu durumda konsol çıktısı doğrudan bir kesme servis rutininde gerçekleştirilmemesi gerektiğinden, çıktıyı RTOS arka plan (daemon) görevi bağlamında (context) gerçekleştirmek için `xTimerPendFunctionCallFromISR()` kullanılır.

Önceki örneklerde olduğu gibi, kesme servis rutini, bir yazılım kesmesini zorlayan (forces) basit bir periyodik görev tarafından tetiklenir (triggered). Bu örnekte kesme her 500 milisaniyede bir üretilir.

```
static uint32_t ulEventBitSettingISR( void )

{

    /* Dize (string) kesme servis rutini içinde yazdırılmaz, bunun yerine

       yazdırılması için RTOS arka plan (daemon) görevine gönderilir.

       Bu nedenle, arka plan görevinden dize yazdırıldığında ISR'nin yığın
çerçevesi

       (stack frame) mevcut olmayacağından, derleyicinin (compiler) dizeyi

       ISR'nin yığınında (stack) tahsis etmemesini sağlamak için static

       olarak bildirilir (declared). */

    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;


    /* Bit 2'nin ayarlanmak üzere olduğunu söyleyen bir mesaj yazdırın.

       Mesajlar bir ISR'den yazdırılamaz, bu nedenle işleve yönelik bir çağrıyı

       RTOS arka plan görevi bağlamında çalışmak üzere bekleterek (pending)

       gerçek çıktıyı RTOS arka plan görevine erteleyin (defer). */

    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask,

                                   ( void * ) pcString,

                                   0,

                                   &xHigherPriorityTaskWoken );
```

```
    /* Olay grubundaki bit 2'yi ayarlayın. */
    xEventGroupSetBitsFromISR( xEventGroup,
                               mainISR_BIT,
                               &xHigherPriorityTaskWoken );

    /* xTimerPendFunctionCallFromISR() ve xEventGroupSetBitsFromISR()
       ikisi de zamanlayıcı komut kuyruğuna (timer command queue) yazar
       ve ikisi de aynı xHigherPriorityTaskWoken değişkenini kullanır.
       Zamanlayıcı komut kuyruğuna yazmak, RTOS arka plan görevinin Engellenmiş
       durumundan çıkmasıyla sonuçlandıysa ve RTOS arka plan görevinin önceliği
       o anda yürütülmekte olan görevin (bu kesmenin kesintiye uğrattığı görevin)
       önceliğinden yüksekse, xHigherPriorityTaskWoken pdTRUE olarak ayarlanmış
       olacaktır.
       xHigherPriorityTaskWoken, portYIELD_FROM_ISR() parametresi olarak
       kullanılır. Eğer xHigherPriorityTaskWoken pdTRUE değerine eşitse,
       portYIELD_FROM_ISR() işlevinin çağrılması bir bağlam geçişi talep
       edecektir. Eğer xHigherPriorityTaskWoken hala pdFALSE ise,
       portYIELD_FROM_ISR() çağrısının hiçbir etkisi olmayacaktır.
       Windows portu tarafından kullanılan portYIELD_FROM_ISR() uygulaması
       bir return ifadesi içerir, bu nedenle bu işlevin açıkça bir değer
       döndürmemesinin nedeni budur. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Liste 9.8 Örnek 9.1'de olay grubundaki bit 2'yi ayarlayan ISR

Liste 9.9, olay grubunda engellenmek (block) için `xEventGroupWaitBits()` işlevini çağıran görevin uygulanmasını göstermektedir. Görev, olay grubunda ayarlanan her bir bit için bir dize yazdırır.

`xEventGroupWaitBits()` işlevinin `xClearOnExit` parametresi `pdTRUE` olarak ayarlanmıştır, bu nedenle `xEventGroupWaitBits()` çağrısının dönmesine (return) neden olan olay biti veya bitleri, `xEventGroupWaitBits()` dönmeden önce otomatik olarak temizlenecektir.

```
static void vEventBitReadingTask( void *pvParameters )
{
    EventBits_t xEventGroupValue;
    const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT |
                                         mainSECOND_TASK_BIT |
                                         mainISR_BIT );

    for( ;; )
    {
        /* Olay grubu içindeki olay bitlerinin ayarlanmasını beklemek için
           engellenin. */
        xEventGroupValue = xEventGroupWaitBits( /* Okunacak olay grubu */
                                                xEventGroup,
                                                /* Test edilecek bitler */
                                                xBitsToWaitFor,
                                                /* Bloktan çıkarma koşulu
                                                   sağlanırsa (met) çıkışta
                                                   bitleri temizleyin (clear) */
                                                pdTRUE,
                                                /* Tüm bitleri beklemeyin. Bu
                                                   parametre ikinci yürütme için
                                                   pdTRUE olarak ayarlanır. */
                                                pdFALSE,
                                                /* Zaman aşımına (time out)
uğramayın. */
                                                portMAX_DELAY );

        /* Ayarlanan her bit için bir mesaj yazdırın. */
        if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )
        {
```

```
            vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );
        }

        if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );
        }

        if( ( xEventGroupValue & mainISR_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );
        }
    }
}
```

Liste 9.9 Örnek 9.1'de olay bitlerinin ayarlanmasını beklemek üzere engellenen görev

`main()` işlevi, çizgeleyiciyi başlatmadan önce olay grubunu ve görevleri oluşturur. Uygulaması için Liste 9.10'a bakın. Olay grubundan okuma yapan görevin önceliği, olay grubuna yazan görevin önceliğinden yüksektir, bu da okuma görevinin bloktan çıkarma koşulu (unblock condition) her sağlandığında (met) okuma görevinin yazma görevini engelleyeceğinden (pre-empt) emin olunmasını sağlar.

```
int main( void )
{
    /* Bir olay grubu kullanılmadan önce oluşturulması gerekir. */
    xEventGroup = xEventGroupCreate();

    /* Olay grubundaki olay bitlerini ayarlayan görevi oluşturun. */
    xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );

    /* Olay grubunda olay bitlerinin ayarlanmasını bekleyen görevi oluşturun. */
    xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );
```

```
    /* Periyodik olarak bir yazılım kesmesi üretmek için kullanılan görevi
       oluşturun. */
    xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );

    /* Yazılım kesmesi için işleyiciyi kurun. Bunu yapmak için gereken
       sözdizimi (syntax), kullanılan FreeRTOS portuna bağlıdır.
       Burada gösterilen sözdizimi yalnızca bu tür kesmelerin simüle
       edildiği FreeRTOS Windows portuyla kullanılabilir. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );

    /* Oluşturulan görevlerin çalışmaya başlaması için çizgeleyiciyi başlatın. */
    vTaskStartScheduler();

    /* Aşağıdaki satıra asla ulaşılmamalıdır. */
    for( ;; );
    return 0;
}
```

Liste 9.10 Örnek 9.1'de olay grubu ve görevlerin oluşturulması

Örnek 9.1, `xEventGroupWaitBits()` işlevinin `xWaitForAllBits` parametresi `pdFALSE` olarak ayarlanmış halde yürütüldüğünde üretilen çıktı Şekil 9.3'te gösterilmektedir. Şekil 9.3'te, `xEventGroupWaitBits()` çağrısındaki `xWaitForAllBits` parametresi `pdFALSE` olarak ayarlandığından, olay grubundan okuma yapan görevin, olay bitlerinden **herhangi biri** ayarlandığında Engellenmiş durumundan çıktığı ve anında yürütüldüğü görülebilir.

```
C:\Windows\system32\cmd.exe - rtosdemo
Bit setting task -       about to set bit 1.
Bit reading task -       event bit 1 was set

Bit setting task -       about to set bit 0.
Bit reading task -       event bit 0 was set

Bit setting task -       about to set bit 1.
Bit reading task -       event bit 1 was set

Bit setting ISR -        about to set bit 2.
Bit reading task -       event bit 2 was set

Bit setting task -       about to set bit 0.
Bit reading task -       event bit 0 was set

Bit setting task -       about to set bit 1.
Bit reading task -       event bit 1 was set

Bit setting ISR -        about to set bit 2.
Bit reading task -       event bit 2 was set

Bit setting task -       about to set bit 0.
Bit reading task -       event bit 0 was set
```

Şekil 9.3 Örnek 9.1'in `xWaitForAllBits` `pdFALSE` olarak ayarlanmış halde yürütüldüğünde üretilen çıktı

Örnek 9.1, `xEventGroupWaitBits()` işlevinin `xWaitForAllBits` parametresi `pdTRUE` olarak ayarlanmış halde yürütüldüğünde üretilen çıktı Şekil 9.4'te gösterilmektedir. Şekil 9.4'te, `xWaitForAllBits` parametresi `pdTRUE` olarak ayarlandığından, olay grubundan okuma yapan görevin ancak olay bitlerinin **üçü de** ayarlandıktan sonra Engellenmiş durumundan çıktığı görülebilir.

```
C:\Windows\system32\cmd.exe - rtosdemo
Bit setting task -       about to set bit 1.
Bit setting task -       about to set bit 0.
Bit setting ISR -        about to set bit 2.
Bit reading task -       event bit 0 was set
Bit reading task -       event bit 1 was set
Bit reading task -       event bit 2 was set

Bit setting task -       about to set bit 1.
Bit setting task -       about to set bit 0.
Bit setting ISR -        about to set bit 2.
Bit reading task -       event bit 0 was set
Bit reading task -       event bit 1 was set
Bit reading task -       event bit 2 was set

Bit setting task -       about to set bit 1.
Bit setting task -       about to set bit 0.
Bit setting task -       about to set bit 1.
Bit setting ISR -        about to set bit 2.
Bit reading task -       event bit 0 was set
Bit reading task -       event bit 1 was set
Bit reading task -       event bit 2 was set

Bit setting task -       about to set bit 0.
Bit setting task -       about to set bit 1.
```

Şekil 9.4 Örnek 9.1'in `xWaitForAllBits` `pdTRUE` olarak ayarlanmış halde yürütüldüğünde üretilen çıktı

9.4 Olay Grubu Kullanarak Görev Senkronizasyonu (Task Synchronization Using an Event Group)

Bazen bir uygulamanın tasarımı iki veya daha fazla görevin birbiriyle senkronize olmasını gerektirir. Örneğin, Görev A'nın bir olay aldığı ve ardından olayın gerektirdiği işlemlerin bir kısmını diğer üç göreve (Görev B, Görev C ve Görev D) devrettiği

(delegates) bir tasarımı düşünün. Görev A, B, C ve D görevlerinin tümü önceki olayı işlemeyi tamamlayana kadar başka bir olay alamıyorsa, dört görevin tümünün birbiriyle senkronize olması gerekecektir. Her görevin senkronizasyon noktası (synchronization point) o görev işlemini tamamladıktan sonra olacaktır ve diğer görevlerin her biri aynısını yapana kadar daha fazla ilerleyemez. Görev A, yalnızca dört görevin tümü senkronizasyon noktasına ulaştıktan sonra başka bir olay alabilir.

Bu tür bir görev senkronizasyonu ihtiyacının daha az soyut (abstract) bir örneği, FreeRTOS+TCP tanıtım (demonstration) projelerinden birinde bulunabilir. Tanıtım, bir TCP soketini iki görev arasında paylaştırır; bir görev sokete veri gönderir ve farklı bir görev aynı soketten veri alır. Başka bir görevin sokete tekrar erişmeye çalışmayacağından emin olana kadar her iki görevin de TCP soketini kapatması güvenli değildir. İki görevden herhangi biri soketi kapatmak isterse, diğer görevi niyetinden (intent) haberdar etmeli ve ardından devam etmeden önce diğer görevin soketi kullanmayı bırakmasını beklemelidir. Sokete veri gönderen görevin soketi kapatmak istediği senaryo Liste 9.11'de gösterilen sözde kodla (pseudo code) açıklanmıştır.

(Not: Yazım sırasında, tek bir FreeRTOS+TCP soketinin görevler arasında paylaşılabilmesinin tek yolu budur.)

Liste 9.11'de gösterilen senaryo önemsizdir (trivial), çünkü birbiriyle senkronize olması gereken sadece iki görev vardır, ancak diğer görevler soketin açık olmasına bağlı işlemler gerçekleştiriyor olsaydı senaryonun nasıl daha karmaşık hale geleceğini ve senkronizasyona katılmak için daha fazla görev gerektireceğini görmek kolaydır.

```
void SocketTxTask( void *pvParameters )

{

    xSocket_t xSocket;

    uint32_t ulTxCount = 0UL;


    for( ;; )

    {

        /* Yeni bir soket oluşturun. Bu görev bu sokete gönderecek (send)

           ve başka bir görev bu soketten veri alacaktır (receive). */

        xSocket = FreeRTOS_socket( ... );


        /* Soketi bağlayın (connect). */

        FreeRTOS_connect( xSocket, ... );


        /* Soketi veri alan göreve göndermek için bir kuyruk kullanın. */
```

```
    xQueueSend( xSocketPassingQueue, &xSocket, portMAX_DELAY );

    /* Soketi kapatmadan önce sokete 1000 mesaj gönderin. */
    for( ulTxCount = 0; ulTxCount < 1000; ulTxCount++ )
    {
        if( FreeRTOS_send( xSocket, ... ) < 0 )
        {
            /* Beklenmeyen hata - döngüden çıkın, sonrasında
               soket kapatılacaktır. */
            break;
        }
    }

    /* Rx (Alma) görevinin, Tx (Gönderme) görevinin soketi
       kapatmak istediğini bilmesini sağlayın. */
    TxTaskWantsToCloseSocket();

    /* Bu Tx görevinin senkronizasyon noktasıdır. Tx görevi
       burada Rx görevinin senkronizasyon noktasına ulaşmasını bekler.
       Rx görevi yalnızca soketi artık kullanmadığında ve soket
       güvenli bir şekilde kapatılabildiğinde senkronizasyon
       noktasına ulaşacaktır. */
    xEventGroupSync( ... );

    /* İki görev de soketi kullanmıyor. Bağlantıyı sonlandırın
       (shut down), ardından soketi kapatın. */
    FreeRTOS_shutdown( xSocket, ... );
    WaitForSocketToDisconnect();
    FreeRTOS_closesocket( xSocket );
}
```

```
}
/*-----------------------------------------------------------*/

void SocketRxTask( void *pvParameters )
{
    xSocket_t xSocket;

    for( ;; )
    {
        /* Tx görevi tarafından oluşturulan ve bağlanan bir
           soketi almak için bekleyin. */
        xQueueReceive( xSocketPassingQueue, &xSocket, portMAX_DELAY );

        /* Tx görevi soketi kapatmak isteyene kadar soketten
           veri almaya devam edin. */
        while( TxTaskWantsToCloseSocket() == pdFALSE )
        {
            /* Alın (receive) ve ardından veriyi işleyin (process). */
            FreeRTOS_recv( xSocket, ... );
            ProcessReceivedData();
        }

        /* Bu Rx görevinin senkronizasyon noktasıdır - buraya
           yalnızca soketi artık kullanmadığında ulaşır ve bu nedenle
           Tx görevinin soketi kapatması güvenlidir. */
        xEventGroupSync( ... );
    }
}
```

Liste 9.11 Soket kapatılmadan önce paylaşılan bir TCP soketinin hiçbir görev tarafından kullanılmadığından emin olmak için birbiriyle senkronize olan iki görev için sözde kod (pseudo code)

Bir senkronizasyon noktası oluşturmak için bir olay grubu kullanılabilir:

- Senkronizasyona katılması gereken her göreve, olay grubu içinde benzersiz (unique) bir olay biti atanır.
- Her görev senkronizasyon noktasına ulaştığında kendi olay bitini ayarlar (sets).
- Kendi olay bitini ayarlayan her görev, diğer tüm senkronizasyon görevlerini temsil eden olay bitlerinin de ayarlanmasını beklemek için olay grubunda engellenir (blocks).

Bununla birlikte, bu senaryoda `xEventGroupSetBits()` ve `xEventGroupWaitBits()` API işlevleri kullanılamaz. Kullanılsalardı, bir bitin ayarlanması (bir görevin senkronizasyon noktasına ulaştığını göstermek için) ve bitlerin test edilmesi (diğer senkronizasyon görevlerinin senkronizasyon noktalarına ulaşıp ulaşmadığını belirlemek için) iki ayrı işlem (operation) olarak gerçekleştirilirdi. Bunun neden bir sorun olacağını görmek için, Görev A, Görev B ve Görev C'nin bir olay grubu kullanarak senkronize olmaya çalıştığı bir senaryoyu düşünün:

1. Görev A ve Görev B senkronizasyon noktasına zaten ulaşmışlardır, bu nedenle olay grubunda kendi olay bitleri ayarlanmıştır (set) ve Görev C'nin olay bitinin de ayarlanmasını beklemek üzere Engellenmiş (Blocked) durumundadırlar.
2. Görev C senkronizasyon noktasına ulaşır ve olay grubundaki bitini ayarlamak için `xEventGroupSetBits()` işlevini kullanır. Görev C'nin biti ayarlanır ayarlanmaz, Görev A ve Görev B Engellenmiş durumundan çıkar ve üç olay bitini de temizler (clear).
3. Görev C daha sonra üç olay bitinin de ayarlanmasını beklemek için `xEventGroupWaitBits()` işlevini çağırır, ancak o zamana kadar üç olay biti de zaten temizlenmiş, Görev A ve Görev B kendi senkronizasyon noktalarından ayrılmıştır ve bu nedenle senkronizasyon başarısız olmuştur.

Bir senkronizasyon noktası oluşturmak için bir olay grubunu başarılı bir şekilde kullanmak için, bir olay bitinin ayarlanması ve ardından olay bitlerinin test edilmesi tek ve kesintiye uğratılamayan (uninterruptable) bir işlem olarak gerçekleştirilmelidir. `xEventGroupSync()` API işlevi bu amaç için sağlanmıştır.

9.4.1 xEventGroupSync() API İşlevi

`xEventGroupSync()` işlevi, iki veya daha fazla görevin birbirleriyle senkronize olmak üzere bir olay grubunu kullanmalarına izin vermek için sağlanmıştır. İşlev, bir görevin bir olay grubunda bir veya daha fazla olay bitini ayarlamasına ve ardından aynı olay grubunda bir dizi olay bitinin ayarlanmasını tek ve kesintiye uğratılamayan bir işlem olarak beklemesine olanak tanır.

`xEventGroupSync()` işlevinin `uxBitsToWaitFor` parametresi çağıran görevin bloktan çıkarma koşulunu belirtir. Eğer `xEventGroupSync()` bloktan çıkarma koşulu sağlandığı

için döndüyse (returned), `uxBitsToWaitFor` tarafından belirtilen olay bitleri `xEventGroupSync()` dönmeden önce tekrar sıfıra temizlenecektir (cleared).

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
                             const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor,
                             TickType_t xTicksToWait );
```

Liste 9.12 `xEventGroupSync()` API işlevi prototipi

`xEventGroupSync()` parametreleri ve dönüş değeri:

- **xEventGroup:** Olay bitlerinin ayarlanacağı (set) ve ardından test edileceği olay grubunun tanıtıcısıdır (handle). Olay grubu tanıtıcısı, olay grubunu oluşturmak için kullanılan `xEventGroupCreate()` çağrısından döndürülmüş olacaktır.
- **`uxBitsToSet`:** Olay grubunda 1 olarak ayarlanacak olay bitini veya olay bitlerini belirten bir bit maskesidir. Olay grubunun değeri, olay grubunun mevcut değeri ile `uxBitsToSet` içinde aktarılan değerin bitsel VEYA (bitwise OR) işlemine tabi tutulmasıyla güncellenir.
  Örnek olarak, `uxBitsToSet` değerini `0x04` (ikili 0100) olarak ayarlamak, olay grubundaki diğer tüm olay bitlerini değiştirmeden bırakırken, olay biti 2'nin ayarlanmasına (eğer zaten ayarlanmamışsa) neden olacaktır.
- **`uxBitsToWaitFor`:** Olay grubunda test edilecek olay bitini veya olay bitlerini belirten bir bit maskesidir.
  Örneğin, çağıran görev olay grubunda olay biti 0, 1 ve 2'nin ayarlanmasını beklemek istiyorsa, `uxBitsToWaitFor` değerini `0x07` (ikili 111) olarak ayarlayın.
- **`xTicksToWait`:** Görevin bloktan çıkarma koşulunun sağlanmasını (met) beklemek için Engellenmiş (Blocked) durumunda kalması gereken maksimum süredir.
  Eğer `xTicksToWait` sıfır ise veya `xEventGroupSync()` çağrıldığı sırada bloktan çıkarma koşulu zaten sağlanmışsa, `xEventGroupSync()` hemen dönecektir.
  Blok süresi tick periyotları (tick periods) cinsinden belirtilir, bu nedenle temsil ettiği mutlak zaman tick frekansına bağlıdır. `pdMS_TO_TICKS()` makrosu, milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için kullanılabilir.
  `FreeRTOSConfig.h` dosyasında `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak görevin süresiz olarak (zaman aşımı olmadan) beklemesine neden olacaktır.
- **Döndürülen Değer:**
  - `xEventGroupSync()` işlevi çağıran görevin bloktan çıkarma koşulu sağlandığı için döndüyse, döndürülen değer çağıran görevin bloktan çıkarma koşulunun sağlandığı andaki (herhangi bir bit otomatik olarak sıfıra temizlenmeden önceki) olay grubunun değeridir. Bu durumda döndürülen değer aynı zamanda çağıran görevin bloktan çıkarma koşulunu da sağlayacaktır.

- `xEventGroupSync()` işlevi `xTicksToWait` parametresi tarafından belirtilen blok süresi dolduğu için döndüyse, döndürülen değer blok süresinin dolduğu andaki olay grubunun değeridir. Bu durumda döndürülen değer çağıran görevin bloktan çıkarma koşulunu sağlamayacaktır.

Örnek 9.2 Görevleri Senkronize Etme (Synchronizing tasks)

Örnek 9.2, tek bir görev uygulamasının (task implementation) üç örneğini (instance) senkronize etmek için `xEventGroupSync()` işlevini kullanır. Görev parametresi (task parameter), görevin `xEventGroupSync()` çağırdığında ayarlayacağı olay bitini her örneğe (instance) aktarmak (pass) için kullanılır.

Görev, `xEventGroupSync()` işlevini çağırmadan önce ve `xEventGroupSync()` çağrısı döndükten sonra bir mesaj yazdırır. Her mesaj bir zaman damgası (time stamp) içerir. Bu, üretilen çıktıda yürütme dizisinin (sequence of execution) gözlemlenmesini sağlar. Tüm görevlerin aynı anda senkronizasyon noktasına ulaşmasını önlemek için sözde rastgele (pseudo random) bir gecikme kullanılır.

Görevin uygulaması için Liste 9.13'e bakın.

```
static void vSyncingTask( void *pvParameters )
{
    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );
    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );
    TickType_t xDelayTime;
    EventBits_t uxThisTasksSyncBit;
    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT |
                                        mainSECOND_TASK_BIT |
                                        mainTHIRD_TASK_BIT );

    /* Bu görevin üç örneği oluşturulur - her görev senkronizasyonda
       farklı bir olay biti kullanır. Kullanılacak olay biti, görev parametresi
       kullanılarak her görev örneğine geçirilir. Bunu uxThisTasksSyncBit
       değişkeninde saklayın. */
    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;

    for( ;; )
    {
```

```
/* Sözde rastgele (pseudo random) bir süre gecikerek bu görevin
   bir eylemi gerçekleştirmesinin biraz zaman aldığını simüle edin.
   Bu, bu görevin üç örneğinin de aynı anda senkronizasyon
   noktasına ulaşmasını önler ve böylece örneğin davranışının
   daha kolay gözlemlenmesini sağlar. */
xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;
vTaskDelay( xDelayTime );

/* Bu görevin senkronizasyon noktasına ulaştığını göstermek için
   bir mesaj yazdırın. pcTaskGetTaskName(), görev oluşturulduğunda
   göreve atanan adı döndüren bir API işlevidir. */
vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );

/* Tüm görevlerin kendi senkronizasyon noktalarına
   ulaşmasını bekleyin. */
xEventGroupSync( /* Senkronize etmek için kullanılan olay grubu. */
                 xEventGroup,
                 /* Senkronizasyon noktasına ulaştığını belirtmek
                    için bu görev tarafından ayarlanan bit. */
                 uxThisTasksSyncBit,
                 /* Beklenecek bitler, senkronizasyonda yer
                    alan her görev için bir bit. */
                 uxAllSyncBits,
                 /* Üç görevin de senkronizasyon noktasına
                    ulaşmasını süresiz olarak (indefinitely) bekleyin. */
                 portMAX_DELAY );

/* Bu görevin senkronizasyon noktasını geçtiğini (passed)
   göstermek için bir mesaj yazdırın. Belirsiz (indefinite) bir
   gecikme kullanıldığından aşağıdaki satır yalnızca tüm görevler
```

```
          kendi senkronizasyon noktalarına ulaştıktan sonra yürütülecektir. */

        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );

    }

}
```

Liste 9.13 Örnek 9.2'de kullanılan görevin uygulaması

`main()` işlevi olay grubunu oluşturur, üç görevin tümünü oluşturur ve ardından çizgeleyiciyi başlatır. Uygulaması için Liste 9.14'e bakın.

```
/* Olay grubundaki olay bitleri için tanımlar. */

#define mainFIRST_TASK_BIT  ( 1UL << 0UL ) /* Olay biti 0, 1. görev tarafından
ayarlanır */

#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Olay biti 1, 2. görev tarafından
ayarlanır */

#define mainTHIRD_TASK_BIT  ( 1UL << 2UL ) /* Olay biti 2, 3. görev tarafından
ayarlanır */



/* Üç görevi senkronize etmek için kullanılan olay grubunu bildirin. */

EventGroupHandle_t xEventGroup;



int main( void )

{

    /* Bir olay grubu kullanılmadan önce oluşturulması gerekir. */

    xEventGroup = xEventGroupCreate();



    /* Görevin üç örneğini oluşturun. Her göreve, daha sonra hangi görevin

       yürütülmekte olduğuna dair görsel bir gösterge vermek için

       yazdırılacak olan farklı bir ad verilir. Görev senkronizasyon

       noktasına ulaştığında kullanılacak olay biti, görev parametresi

       kullanılarak göreve geçirilir. */

    xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL );

    xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );
```

```
    xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );


    /* Oluşturulan görevlerin çalışmaya başlaması için çizgeleyiciyi başlatın. */
    vTaskStartScheduler();


    /* Her zaman olduğu gibi aşağıdaki satıra asla ulaşılmamalıdır. */
    for( ;; );
    return 0;
}
```

Liste 9.14 Örnek 9.2'de kullanılan `main()` işlevi

Örnek 9.2 çalıştırıldığında üretilen çıktı Şekil 9.5'te gösterilmektedir. Her görevin senkronizasyon noktasına farklı (sözde rastgele) bir zamanda ulaşmasına rağmen, her görevin senkronizasyon noktasından aynı anda (yani son görevin senkronizasyon noktasına ulaştığı zamanda) çıktığı görülebilir.

(Not: Şekil 9.5, örneğin gerçek zamanlı - true real time - davranış sağlamayan FreeRTOS Windows portunda çalıştığını göstermektedir. Özellikle konsola yazdırmak için Windows sistem çağrılarını kullanırken ve bu nedenle bazı zamanlama farklılıkları - timing variation - gösterecektir.)

```
C:\Windows\system32\cmd.exe - rtosdemo
At time 211664: Task 1 reached sync point
At time 211664: Task 1 exited sync point
At time 211664: Task 2 exited sync point
At time 211664: Task 3 exited sync point
At time 212702: Task 2 reached sync point
At time 214400: Task 1 reached sync point
At time 215439: Task 3 reached sync point
At time 215439: Task 3 exited sync point
At time 215439: Task 2 exited sync point
At time 215440: Task 1 exited sync point
At time 217671: Task 2 reached sync point
At time 218622: Task 1 reached sync point
At time 219402: Task 3 reached sync point
At time 219402: Task 3 exited sync point
At time 219402: Task 2 exited sync point
At time 219402: Task 1 exited sync point
At time 220189: Task 2 reached sync point
At time 222656: Task 3 reached sync point
At time 222673: Task 1 reached sync point
At time 222673: Task 1 exited sync point
At time 222673: Task 2 exited sync point
At time 222673: Task 3 exited sync point
At time 223252: Task 1 reached sync point
At time 223682: Task 3 reached sync point
```

Şekil 9.5 Örnek 9.2 yürütüldüğünde üretilen çıktı

Bölüm 10

Görev Bildirimleri (Task Notifications)

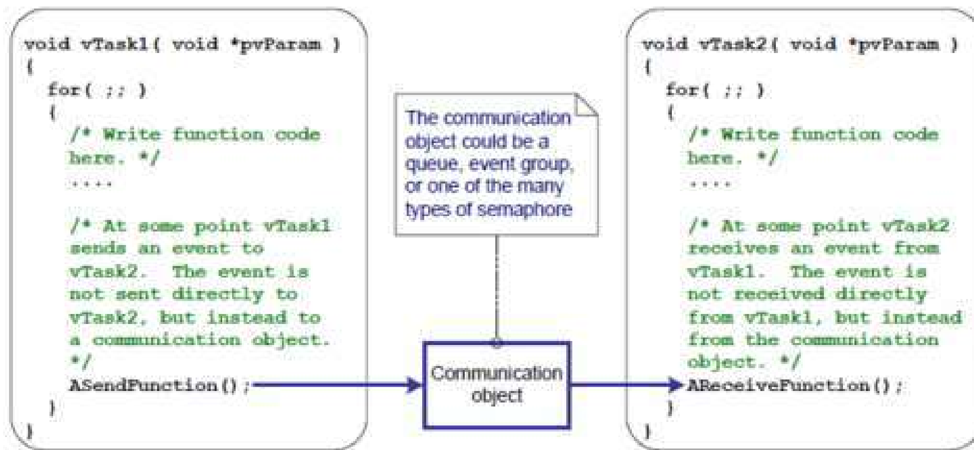
10.1 Giriş

FreeRTOS uygulamaları, sistem işlevselliğini toplu (collectively) olarak sağlamak için birbirleriyle iletişim kuran bir dizi bağımsız görev olarak yapılandırılır. Görev bildirimleri (task notifications), bir görevin doğrudan başka bir görevi bilgilendirmesine olanak tanıyan verimli (efficient) bir mekanizmadır.

10.1.1 Aracı (Intermediary) Nesneler Üzerinden İletişim Kurma

Bu kitapta görevlerin birbirleriyle iletişim kurabileceği çeşitli yollar zaten açıklanmıştır. Şimdiye kadar açıklanan yöntemler bir iletişim nesnesinin (communication object) oluşturulmasını gerektiriyordu. İletişim nesnelerine örnek olarak kuyruklar (queues), olay grupları (event groups) ve çeşitli semafor türleri verilebilir.

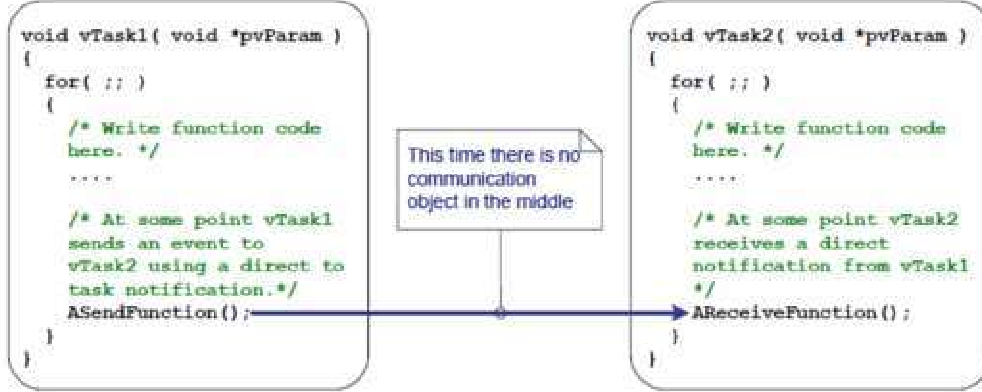
Bir iletişim nesnesi kullanıldığında, olaylar ve veriler doğrudan alıcı bir göreve veya alıcı bir ISR'ye (Kesme Hizmet Rutini) gönderilmez, bunun yerine iletişim nesnesine gönderilir. Benzer şekilde, görevler ve ISR'ler olayları ve verileri doğrudan olay veya veriyi gönderen görev veya ISR'den almak yerine iletişim nesnesinden alırlar. Bu durum Şekil 10.1'de gösterilmiştir.



Şekil 10.1 Bir görevden diğerine olay göndermek için kullanılan bir iletişim nesnesi

10.1.2 Görev Bildirimleri — Doğrudan Görevle İletişim

'Görev Bildirimleri' (Task Notifications), ayrı bir iletişim nesnesine ihtiyaç duymadan görevlerin diğer görevlerle etkileşime girmesine ve ISR'lerle senkronize olmasına olanak tanır. Bir görev bildirimi kullanarak, bir görev veya ISR doğrudan alıcı (receiving) göreve bir olay gönderebilir. Bu durum Şekil 10.2'de gösterilmiştir.



Şekil 10.2 Bir görevden diğerine doğrudan olay göndermek için kullanılan görev bildirimi

Görev bildirimi işlevselliği isteğe bağlıdır. Görev bildirimi işlevselliğini dahil etmek için `FreeRTOSConfig.h` dosyasında `configUSE_TASK_NOTIFICATIONS` değerini 1 olarak ayarlayın.

`configUSE_TASK_NOTIFICATIONS` 1 olarak ayarlandığında, her görevin 'Beklemede' (Pending) veya 'Beklemede Değil' (Not-Pending) olabilen en az bir 'Bildirim Durumu' (Notification State) ve 32 bitlik işaretsiz (unsigned) bir tamsayı olan bir 'Bildirim Değeri' (Notification Value) vardır. Bir görev bildirim aldığında, bildirim durumu beklemede olarak ayarlanır. Bir görev bildirim değerini okuduğunda, bildirim durumu beklemede değil olarak ayarlanır. Eğer `configTASK_NOTIFICATION_ARRAY_ENTRIES` 1'den büyük bir değere ayarlanmışsa, dizin (index) ile tanımlanan birden fazla Bildirim durumu ve değeri vardır.

Bir görev, bildirim durumunun beklemede olmasını (pending) isteğe bağlı bir zaman aşımı (time out) ile Engellenmiş (Blocked) durumunda bekleyebilir.

10.1.3 Kapsam (Scope)

Bu bölümde aşağıdakiler tartışılmaktadır:

- Bir görevin bildirim durumu ve bildirim değeri.
- Bir görev bildiriminin semafor gibi bir iletişim nesnesi yerine nasıl ve ne zaman kullanılabileceği.
- İletişim nesnesi yerine görev bildirimi kullanmanın avantajları.

10.2 Görev Bildirimleri; Avantajlar ve Sınırlamalar (Benefits and Limitations)

10.2.1 Görev Bildirimlerinin Performans Avantajları

Bir göreve olay veya veri göndermek için görev bildirimi kullanmak, eşdeğer (equivalent) bir işlemi gerçekleştirmek için kuyruk, semafor veya olay grubu kullanmaktan önemli ölçüde (significantly) daha hızlıdır.

10.2.2 Görev Bildirimlerinin RAM Ayak İzi (Footprint) Avantajları

Benzer şekilde, bir göreve olay veya veri göndermek için bir görev bildirimi kullanmak, eşdeğer bir işlemi gerçekleştirmek için kuyruk, semafor veya olay grubu kullanmaktan önemli ölçüde daha az RAM gerektirir. Bunun nedeni, her bir iletişim nesnesinin (kuyruk, semafor veya olay grubu) kullanılabilmesi için önce oluşturulması gerekmesidir, oysa görev bildirimi işlevini etkinleştirmenin sabit bir genel yükü (fixed overhead) vardır. Görev bildirimleri için RAM maliyeti, görev başına `configTASK_NOTIFICATION_ARRAY_ENTRIES * 8` bayttır. `configTASK_NOTIFICATION_ARRAY_ENTRIES` için varsayılan değer 1'dir, bu da görev bildirimleri için varsayılan boyutun görev başına 8 bayt olmasını sağlar.

10.2.3 Görev Bildirimlerinin Sınırlamaları (Limitations)

Görev bildirimleri daha hızlıdır ve iletişim nesnelerinden daha az RAM kullanır, ancak görev bildirimleri tüm senaryolarda kullanılamaz. Bu bölüm, bir görev bildiriminin kullanılamayacağı senaryoları belgeler:

- **Bir ISR'ye olay veya veri gönderme**
  İletişim nesneleri, bir ISR'den bir göreve ve bir görevden bir ISR'ye olay ve veri göndermek için kullanılabilir.
  Görev bildirimleri bir ISR'den bir göreve olay ve veri göndermek için kullanılabilir, ancak bir görevden bir ISR'ye olay veya veri göndermek için kullanılamazlar.
- **Birden fazla alıcı görevi etkinleştirme**
  Bir iletişim nesnesine, tanıtıcısını (kuyruk tanıtıcısı, semafor tanıtıcısı veya olay grubu tanıtıcısı olabilir) bilen herhangi bir görev veya ISR tarafından erişilebilir. Herhangi bir sayıda görev ve ISR, belirli bir iletişim nesnesine gönderilen olayları veya verileri işleyebilir.
  Görev bildirimleri doğrudan alıcı göreve gönderilir, bu nedenle yalnızca bildirimin gönderildiği görev tarafından işlenebilirler. Bununla birlikte, pratik durumlarda bu nadiren bir sınırlamadır çünkü birden fazla görev ve ISR'nin aynı iletişim nesnesine veri göndermesi yaygın olsa da, birden fazla görev ve ISR'nin aynı iletişim nesnesinden veri alması nadirdir.
- **Birden fazla veri öğesini arabelleğe alma (Buffering)**
  Kuyruk, aynı anda birden fazla veri öğesini tutabilen bir iletişim nesnesidir. Kuyruğa gönderilen ancak henüz kuyruktan alınmayan (received) veriler kuyruk nesnesi içinde arabelleğe alınır.

Görev bildirimleri, alıcı görevin bildirim değerini güncelleyerek göreve veri gönderir. Bir görevin bildirim değeri aynı anda yalnızca bir değer tutabilir.

- **Aynı anda birden fazla göreve yayın yapma (Broadcasting)**
Olay grubu, bir olayı aynı anda birden fazla göreve göndermek için kullanılabilecek bir iletişim nesnesidir.
Görev bildirimleri doğrudan alıcı göreve gönderilir, bu nedenle yalnızca alıcı görev tarafından işlenebilir.

- **Bir gönderimin tamamlanması için Engellenmiş durumunda bekleme**
Bir iletişim nesnesi geçici olarak daha fazla veri veya olayın yazılamayacağı bir durumdaysa (örneğin, bir kuyruk dolduğunda kuyruğa daha fazla veri gönderilemez), nesneye yazmaya çalışan görevler yazma işlemlerinin tamamlanmasını beklemek için isteğe bağlı olarak Engellenmiş (Blocked) durumuna girebilir.
Bir görev, zaten bekleyen (pending) bir bildirimi olan bir göreve görev bildirimi göndermeye çalışırsa, gönderen görevin alıcı görevin bildirim durumunu sıfırlamasını (reset) Engellenmiş durumunda beklemesi mümkün değildir.
Görüleceği gibi, bu bir görev bildiriminin kullanıldığı pratik durumlarda nadiren bir sınırlamadır.

10.3 Görev Bildirimlerini Kullanma

10.3.1 Görev Bildirim API Seçenekleri

Görev bildirimleri, ikili semafor (binary semaphore), sayma semaforu (counting semaphore), olay grubu (event group) ve bazen bir kuyruk (queue) yerine kullanılabilen çok güçlü bir özelliktir. Bu geniş kullanım senaryoları, bir görev bildirimi göndermek için `xTaskNotify()` API işlevi ve bir görev bildirimi almak için `xTaskNotifyWait()` API işlevi kullanılarak elde edilebilir.

Bununla birlikte, vakaların çoğunda `xTaskNotify()` ve `xTaskNotifyWait()` API işlevlerinin sağladığı tam esneklik (flexibility) gerekli değildir ve daha basit işlevler yeterli olacaktır. Bu nedenle, `xTaskNotifyGive()` API işlevi `xTaskNotify()` işlevine daha basit ancak daha az esnek bir alternatif olarak sunulmuştur ve `ulTaskNotifyTake()` API işlevi `xTaskNotifyWait()` işlevine daha basit ancak daha az esnek bir alternatif olarak sunulmuştur.

Görev bildirim sistemi tek bir bildirim olayıyla sınırlı değildir. `configTASK_NOTIFICATION_ARRAY_ENTRIES` yapılandırma parametresi varsayılan olarak 1'e ayarlıdır. 1'den büyük bir değere ayarlanırsa, her görevin içinde bir bildirim dizisi oluşturulur. Bu, bildirimlerin dizin (index) tarafından yönetilmesini sağlar. Her görev bildirimi API işlevinin dizinli (indexed) bir sürümü vardır. Dizinlenmemiş sürümün kullanılması `notification[0]` öğesine (dizideki ilk öğe) erişilmesine neden olur. Her API işlevinin dizinli sürümü `Indexed` eki ile tanımlanır, böylece `xTaskNotify` işlevi `xTaskNotifyIndexed` olur. Basitlik sağlamak amacıyla bu kitap boyunca her bir işlevin yalnızca dizinlenmemiş (non-indexed) sürümleri kullanılacaktır.

Görev bildirimi API'leri, her API işlevi türünün temel Jenerik sürümlerine (Generic versions) çağrı yapan makrolar olarak uygulanır. Basitlik için API makroları bu kitap boyunca işlev olarak adlandırılacaktır.

10.3.1.1 API İşlevlerinin Tam Listesi [27]

- `xTaskNotifyGive`
- `xTaskNotifyGiveIndexed`
- `vTaskNotifyGiveFromISR`
- `vTaskNotifyGiveIndexedFromISR`
- `ulTaskNotifyTake`
- `ulTaskNotifyTakeIndexed`
- `xTaskNotify`
- `xTaskNotifyIndexed`
- `xTaskNotifyWait`
- `xTaskNotifyWaitIndexed`
- `xTaskNotifyStateClear`
- `xTaskNotifyStateClearIndexed`
- `ulTaskNotifyValueClear`
- `ulTaskNotifyValueClearIndexed`
- `xTaskNotifyAndQueryIndexedFromISR`
- `xTaskNotifyAndQueryFromISR`
- `xTaskNotifyFromISR`
- `xTaskNotifyIndexedFromISR`
- `xTaskNotifyAndQuery`
- `xTaskNotifyAndQueryIndexed`

(27): Bu işlevler aslında makrolar olarak uygulanmaktadır.

Not: FromISR işlevleri bildirimleri almak (receive) için mevcut değildir, çünkü bir bildirim her zaman bir göreve gönderilir ve kesmeler (interrupts) herhangi bir görevle ilişkilendirilmez.

10.3.2 xTaskNotifyGive() API İşlevi

`xTaskNotifyGive()`, bir bildirimi doğrudan bir göreve gönderir ve alıcı görevin bildirim değerini bir artırır (increments). `xTaskNotifyGive()` çağrısı, daha önce beklemede değilse, alıcı görevin bildirim durumunu beklemede (pending) olarak ayarlar.

`xTaskNotifyGive()` API işlevi, bir görev bildiriminin ikili (binary) veya sayma (counting) semaforuna daha hafif (lighter weight) ve daha hızlı bir alternatif olarak kullanılmasına izin vermek için sağlanmıştır.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

```
BaseType_t xTaskNotifyGiveIndexed( TaskHandle_t xTaskToNotify, UBaseType_t
uxIndexToNotify );
```

Liste 10.1 `xTaskNotifyGive()` API işlevi prototipi

`xTaskNotifyGive()`/`xTaskNotifyGiveIndexed`() parametreleri ve dönüş değeri:

- **`xTaskToNotify`:** Bildirimin gönderildiği görevin tanıtıcısıdır (handle). Görevlere ait tanıtıcıların elde edilmesiyle ilgili bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine bakın.
- **uxIndexToNotify:** Dizideki (array) dizindir.
- **Döndürülen Değer:** `xTaskNotifyGive()`, `xTaskNotify()` öğesini çağıran bir makrodur. Makro tarafından `xTaskNotify()` işlevine aktarılan parametreler, mümkün olan tek dönüş değerinin `pdPASS` olacağı şekilde ayarlanmıştır. `xTaskNotify()` bu kitabın ilerleyen bölümlerinde açıklanmıştır.

10.3.3 vTaskNotifyGiveFromISR() API İşlevi

`vTaskNotifyGiveFromISR()`, bir kesme hizmet rutininde (interrupt service routine - ISR) kullanılabilen `xTaskNotifyGive()` işlevinin bir sürümüdür.

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,

                             BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 10.2 `vTaskNotifyGiveFromISR()` API işlevi prototipi

`vTaskNotifyGiveFromISR()` parametreleri ve dönüş değeri:

- **`xTaskToNotify`:** Bildirimin gönderildiği görevin tanıtıcısıdır. Görevlere ait tanıtıcıların elde edilmesiyle ilgili bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine bakın.
- **`pxHigherPriorityTaskWoken`:** Bildirimin gönderildiği görev bir bildirim almak üzere Engellenmiş (Blocked) durumunda bekliyorsa, bildirimin gönderilmesi görevin Engellenmiş durumundan çıkmasına (leave) neden olur.
  `vTaskNotifyGiveFromISR()` işlevinin çağrılması bir görevin Engellenmiş durumundan çıkmasına neden olursa ve bloktan çıkan görev o anda yürütülen görevin (kesintiye uğrayan görev) önceliğinden daha yüksek bir önceliğe sahipse, dahili olarak `vTaskNotifyGiveFromISR()`, `*pxHigherPriorityTaskWoken` değerini `pdTRUE` olarak ayarlayacaktır.
  `vTaskNotifyGiveFromISR()` bu değeri `pdTRUE` olarak ayarlarsa, kesmeden (interrupt) çıkılmadan önce bir bağlam anahtarı (context switch) gerçekleştirilmelidir. Bu, kesmenin doğrudan en yüksek öncelikli Hazır (Ready) durumundaki göreve dönmesini (return) sağlayacaktır.
  Kesme için güvenli tüm API işlevlerinde (interrupt safe API functions) olduğu gibi, `pxHigherPriorityTaskWoken` parametresi kullanılmadan önce `pdFALSE` olarak ayarlanmalıdır.

10.3.4 ulTaskNotifyTake() API İşlevi

`ulTaskNotifyTake()`, bir görevin bildirim değerinin sıfırdan büyük olması için Engellenmiş (Blocked) durumunda beklemesine olanak tanır ve geri dönmeden (returns) önce görevin bildirim değerini ya bir azaltır (decrements) ya da temizler (clears).

`ulTaskNotifyTake()` API işlevi, bir görev bildiriminin ikili (binary) veya sayma (counting) semaforuna daha hafif ve daha hızlı bir alternatif olarak kullanılmasına izin vermek için sağlanmıştır.

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait
);
```

Liste 10.3 `ulTaskNotifyTake()` API işlevi prototipi

`ulTaskNotifyTake()` parametreleri ve dönüş değeri:

- **`xClearCountOnExit`:** `xClearCountOnExit` `pdTRUE` olarak ayarlanırsa, `ulTaskNotifyTake()` çağrısı dönmeden önce çağıran görevin bildirim değeri sıfırlanacaktır (cleared to zero).
  Eğer `xClearCountOnExit` `pdFALSE` olarak ayarlanırsa ve çağıran görevin bildirim değeri sıfırdan büyükse, `ulTaskNotifyTake()` çağrısı dönmeden önce çağıran görevin bildirim değeri azaltılacaktır (decremented).
- **`xTicksToWait`:** Çağıran görevin bildirim değerinin sıfırdan büyük olmasını beklemek için Engellenmiş durumunda kalması gereken maksimum süredir.
  Blok süresi tick periyotları (tick periods) cinsinden belirtilir, bu nedenle temsil ettiği mutlak zaman tick frekansına bağlıdır. `pdMS_TO_TICKS()` makrosu, milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için kullanılabilir.
  `FreeRTOSConfig.h` dosyasında `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak görevin süresiz olarak (zaman aşımı olmadan) beklemesine neden olacaktır.
- **Döndürülen Değer:** Döndürülen değer, çağıran görevin bildirim değerinin `xClearCountOnExit` parametresinin değerine göre sıfırlanmadan veya azaltılmadan önceki halidir.
  Bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse) ve dönüş değeri sıfır değilse, çağıran görev bildirim değerinin sıfırdan büyük olmasını beklemek için Engellenmiş durumuna yerleştirilmiş (placed) ve bildirim değeri blok süresi dolmadan önce güncellenmiş olabilir.
  Bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse) ve dönüş değeri sıfırsa, çağıran görev bildirim değerinin sıfırdan büyük olmasını beklemek üzere Engellenmiş durumuna yerleştirilmiş (placed), ancak belirtilen blok süresi bu gerçekleşmeden önce dolmuştur.

Örnek 10.1 Semafor yerine görev bildirimi kullanma, yöntem 1

Örnek 7.1, bir görevi bir kesme hizmet rutininin (interrupt service routine) içinden bloktan çıkarmak (unblock) için bir ikili semafor kullandı - etkili bir şekilde görevi kesme (interrupt) ile senkronize etti. Bu örnek, Örnek 7.1'in işlevselliğini tekrarlar (replicates), ancak ikili semafor yerine doğrudan göreve bildirim (direct to task notification) kullanır.

Liste 10.4, kesme ile senkronize edilen görevin uygulamasını (implementation) göstermektedir. Örnek 7.1'de kullanılan `xSemaphoreTake()` çağrısının yerini `ulTaskNotifyTake()` çağrısı almıştır.

`ulTaskNotifyTake()` `xClearCountOnExit` parametresi `pdTRUE` olarak ayarlanmıştır, bu da alıcı görevin bildirim değerinin `ulTaskNotifyTake()` dönmeden önce sıfıra temizlenmesiyle (cleared) sonuçlanır. Bu nedenle, `ulTaskNotifyTake()` işlevine yapılan her çağrı arasında halihazırda mevcut olan tüm olayların işlenmesi (process) gerekir. Örnek 7.1'de, ikili bir semafor kullanıldığı için, bekleyen olayların sayısının her zaman pratik olmayan donanımdan (hardware) belirlenmesi gerekiyordu. Örnek 10.1'de, bekleyen olayların sayısı `ulTaskNotifyTake()` işlevinden döndürülür.

`ulTaskNotifyTake` çağrıları arasında meydana gelen kesme olayları görevin bildirim değerinde kilitlenir (latched) ve çağıran görevin halihazırda bekleyen (pending) bildirimleri varsa `ulTaskNotifyTake()` çağrıları hemen geri döner.

```
/* Periyodik görevin yazılım kesmeleri oluşturma hızı. */

const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );



static void vHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime, olaylar arasındaki maksimum beklenen

       süreden biraz daha uzun olacak şekilde ayarlanmıştır. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS(
10 );

    uint32_t ulEventsToProcess;



    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü

       içinde uygulanır. */

    for( ;; )

    {

        /* Kesme hizmet rutininden (ISR) doğrudan bu göreve

           gönderilen bir bildirimi almak için bekleyin. */
```

```
        ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );


        if( ulEventsToProcess != 0 )
        {
            /* Buraya ulaşmak için en az bir olay (event) meydana
               gelmiş olmalıdır. Bekleyen tüm olaylar işlenene kadar
               burada döngüye girin (bu durumda, sadece her olay
               için bir mesaj yazdırın). */
            while( ulEventsToProcess > 0 )
            {
                vPrintString( "Handler task - Processing event.\r\n" );
                ulEventsToProcess--;
            }
        }
        else
        {
            /* İşlevin bu kısmına ulaşılırsa, beklenen süre içinde
               bir kesme (interrupt) gelmemiştir ve (gerçek bir uygulamada)
               bazı hata kurtarma (error recovery) işlemleri
               gerçekleştirmek gerekebilir. */
        }
    }
}
```

Liste 10.4 Örnek 10.1'de kesme işleminin (interrupt processing) ertelendiği (deferred) (kesme ile senkronize olan) görevin uygulaması

Yazılım kesmeleri oluşturmak için kullanılan periyodik görev, kesme oluşturulmadan önce ve kesme oluşturulduktan sonra birer mesaj yazdırır. Bu, yürütme dizisinin (sequence of execution) üretilen çıktıda gözlemlenmesini sağlar.

Liste 10.5 kesme işleyicisini (interrupt handler) göstermektedir. Bu, kesme işlemenin (interrupt handling) ertelendiği göreve doğrudan bir bildirim göndermekten başka çok az şey yapar.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    /* xHigherPriorityTaskWoken parametresi pdFALSE olarak
       ilklendirilmelidir, çünkü bir bağlam anahtarı (context switch) gerekirse
       kesme (interrupt) güvenli API işlevi içinde pdTRUE olarak ayarlanacaktır.
*/
    xHigherPriorityTaskWoken = pdFALSE;

    /* Kesme işleminin ertelendiği göreve doğrudan bir
       bildirim (notification) gönderin. */
    vTaskNotifyGiveFromISR( /* Bildirimin gönderildiği görevin tanıtıcısı.
                               Tanıtıcı (handle) görev oluşturulduğunda
                               kaydedilmişti. */
                            xHandlerTask,
                            /* xHigherPriorityTaskWoken olağan
                               şekilde kullanılır. */
                            &xHigherPriorityTaskWoken );

    /* xHigherPriorityTaskWoken değerini portYIELD_FROM_ISR() içine
       geçirin. vTaskNotifyGiveFromISR() içinde xHigherPriorityTaskWoken
       pdTRUE olarak ayarlanmışsa portYIELD_FROM_ISR() öğesini
       çağırmak bir bağlam anahtarı (context switch) isteyecektir.
       xHigherPriorityTaskWoken hala pdFALSE ise portYIELD_FROM_ISR()
       çağrısının hiçbir etkisi olmayacaktır. Windows portu tarafından kullanılan
       portYIELD_FROM_ISR() uygulaması (implementation) bir dönüş (return)
       ifadesi içerir, bu nedenle bu işlev açıkça (explicitly) bir değer
döndürmez. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

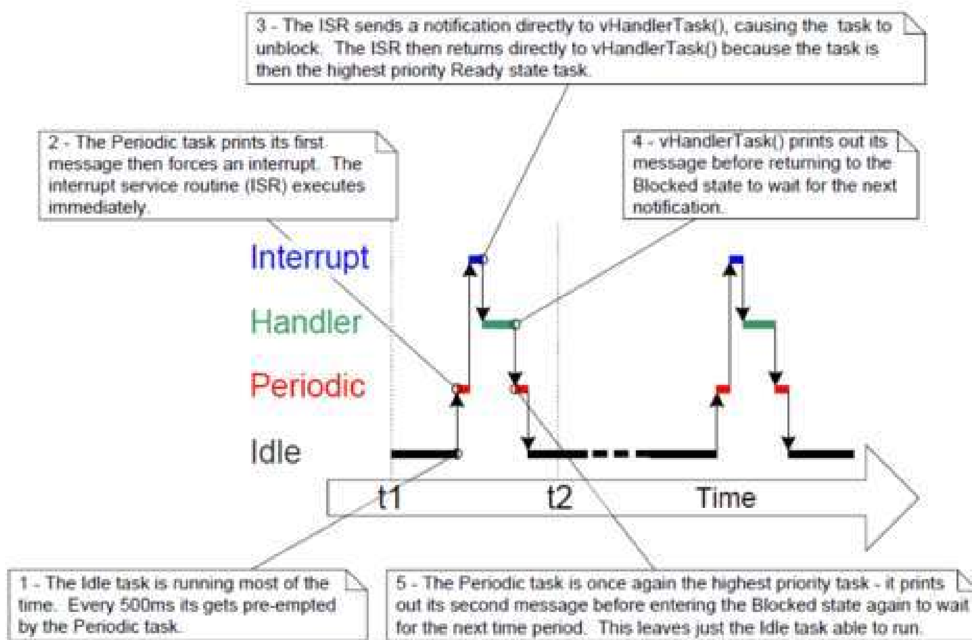
```
}
```

Liste 10.5 Örnek 10.1'de kullanılan kesme hizmet rutininin (ISR) uygulaması

Örnek 10.1 yürütüldüğünde üretilen çıktı Şekil 10.3'te gösterilmiştir. Beklendiği gibi, Örnek 7.1 yürütüldüğünde üretilenle aynıdır. `vHandlerTask()` kesme (interrupt) üretilir üretilmez Çalışıyor (Running) durumuna girer, bu nedenle görevden gelen çıktı periyodik görev tarafından üretilen çıktıyı böler (splits). Daha fazla açıklama Şekil 10.4'te verilmiştir.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
```

Şekil 10.3 Örnek 10.1 (7.1'e benzer) yürütüldüğünde üretilen çıktı



Şekil 10.4 Örnek 10.1 yürütüldüğünde yürütme dizisi (sequence of execution)

Örnek 10.2 Semafor yerine görev bildirimi kullanma, yöntem 2

Örnek 10.1'de `ulTaskNotifyTake()` `xClearOnExit` parametresi `pdTRUE` olarak ayarlanmıştı. Örnek 10.2, `ulTaskNotifyTake()` `xClearOnExit` parametresinin `pdFALSE` olarak ayarlandığı durumu (behavior) göstermek için Örnek 10.1'i biraz değiştirir.

`xClearOnExit` `pdFALSE` olduğunda, `ulTaskNotifyTake()` öğesini çağırmak, sıfırlamak yerine yalnızca çağıran görevin bildirim değerini azaltır (reduce by one). Bu nedenle bildirim sayısı (notification count), meydana gelen olay sayısı ile işlenen (processed) olay sayısı arasındaki farktır. Bu, `vHandlerTask()` yapısının iki yolla basitleştirilmesine (simplified) olanak tanır:

1. İşlenmeyi bekleyen olayların sayısı bildirim değerinde tutulur, bu nedenle yerel bir değişkende tutulmasına gerek yoktur.
2. Her bir `ulTaskNotifyTake()` çağrısı arasında sadece bir olay işlemek (process) gerekir.

Örnek 10.2'de kullanılan `vHandlerTask()` uygulaması Liste 10.6'da gösterilmektedir.

```
static void vHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime, olaylar arasındaki maksimum beklenen

       süreden biraz daha uzun olacak şekilde ayarlanmıştır. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS(
10 );



    /* Çoğu görevde olduğu gibi, bu görev de sonsuz bir döngü

       içinde uygulanır. */

    for( ;; )

    {

        /* Kesme hizmet rutininden (ISR) doğrudan bu göreve gönderilen

           bir bildirimi almak için bekleyin. xClearCountOnExit parametresi

           artık pdFALSE'dir, bu nedenle görevin bildirim değeri sıfıra

           temizlenmek yerine ulTaskNotifyTake() tarafından azaltılacaktır

           (decremented). */

        if( ulTaskNotifyTake( pdFALSE, xMaxExpectedBlockTime ) != 0 )
```

```
        {
            /* Buraya ulaşmak için bir olay (event) meydana gelmiş olmalıdır.
               Olayı işleyin (bu durumda sadece bir mesaj yazdırın). */
            vPrintString( "Handler task - Processing event.\r\n" );
        }
        else
        {
            /* İşlevin bu kısmına ulaşılırsa, beklenen süre içinde bir kesme
               (interrupt) gelmemiştir ve (gerçek bir uygulamada) bazı hata
               kurtarma (error recovery) işlemleri gerçekleştirmek gerekebilir. */
        }
    }
}
```

Liste 10.6 Örnek 10.2'de kesme işleminin (interrupt processing) ertelendiği (kesme ile senkronize olan) görevin uygulaması

Gösterim (demonstration) amacıyla, kesme hizmet rutini ayrıca kesme başına (per interrupt) birden fazla görev bildirimi gönderecek şekilde değiştirilmiş ve böylece yüksek frekansta (high frequency) meydana gelen birden fazla kesmeyi (multiple interrupts) simüle etmiştir. Örnek 10.2'de kullanılan kesme hizmet rutininin uygulaması Liste 10.7'de gösterilmektedir.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* İşleyici (handler) görevine birden fazla kez bildirim gönderin.
       İlk 'give' görevi bloktan çıkaracaktır (unblock), sonraki 'gives' alıcı
       görevin bildirim değerinin olayları saymak (kilitlemek - latch) için
       kullanıldığını göstermek içindir - bu da görevin her olayı
```

```
        sırayla işlemesini sağlar. */

    vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken );

    vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken );

    vTaskNotifyGiveFromISR( xHandlerTask, &xHigherPriorityTaskWoken );


    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Liste 10.7 Örnek 10.2'de kullanılan kesme hizmet rutininin (ISR) uygulaması

Örnek 10.2 yürütüldüğünde üretilen çıktı Şekil 10.5'te gösterilmektedir. Görülebileceği gibi, `vHandlerTask()` her kesme (interrupt) üretildiğinde üç olayın tümünü işler.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Perodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

Şekil 10.5 Örnek 10.2 yürütüldüğünde üretilen çıktı

10.3.5 xTaskNotify() ve xTaskNotifyFromISR() API İşlevleri

`xTaskNotify()`, alıcı görevin bildirim değerini aşağıdaki yollardan (ways) herhangi biriyle güncellemek için kullanılabilen `xTaskNotifyGive()` işlevinin daha yetenekli bir sürümüdür:

- Alıcı görevin bildirim değerini artırın (increment); bu durumda `xTaskNotify()` işlevi `xTaskNotifyGive()` işlevine eşdeğerdir.
- Alıcı görevin bildirim değerinde bir veya daha fazla bit ayarlayın (set). Bu, bir görevin bildirim değerinin olay grubuna (event group) daha hafif ve daha hızlı bir alternatif olarak kullanılmasına olanak tanır.
- Alıcı görevin bildirim değerine tamamen yeni bir sayı yazın, ancak yalnızca alıcı görev son güncellendiğinden (updated) beri bildirim değerini okumuşsa.

Bu, bir görevin bildirim değerinin bir (1) uzunluğunda bir kuyruk (queue) tarafından sağlanana benzer işlevsellik sağlamasına olanak tanır.

- Alıcı görev son güncellendiğinden beri bildirim değerini okumamış olsa bile, alıcı görevin bildirim değerine tamamen yeni bir sayı yazın. Bu, bir görevin bildirim değerinin `xQueueOverwrite()` API işlevi tarafından sağlanan işlevselliğe benzer bir işlevsellik sağlamasına olanak tanır. Ortaya çıkan davranış bazen 'posta kutusu' (mailbox) olarak adlandırılır.

`xTaskNotify()`, `xTaskNotifyGive()` işlevinden daha esnek ve güçlüdür ve bu ekstra esneklik ve güç nedeniyle kullanımı biraz daha karmaşıktır (complex).

`xTaskNotifyFromISR()`, `xTaskNotify()` işlevinin bir kesme hizmet rutininde (ISR) kullanılabilen bir sürümüdür ve bu nedenle ek bir `pxHigherPriorityTaskWoken` parametresine sahiptir.

`xTaskNotify()` çağrısı, daha önce beklemede değilse, her zaman alıcı görevin bildirim durumunu beklemede (pending) olarak ayarlayacaktır.

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );


BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify,
                               uint32_t ulValue,
                               eNotifyAction eAction,
                               BaseType_t *pxHigherPriorityTaskWoken );
```

Liste 10.8 `xTaskNotify()` ve `xTaskNotifyFromISR()` API işlevleri için prototipler

`xTaskNotify()` parametreleri ve dönüş değeri:

- **`xTaskToNotify`:** Bildirimin gönderildiği görevin tanıtıcısıdır (handle). Görevlere ait tanıtıcıların elde edilmesiyle ilgili bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine bakın.
- **`ulValue`:** `ulValue`'nun nasıl kullanıldığı `eNotifyAction` değerine bağlıdır. Aşağıya bakınız.
- **`eAction`:** Alıcı görevin bildirim değerinin nasıl güncelleneceğini (update) belirten numaralandırılmış (enumerated) bir tür. Aşağıya bakınız.
- **Döndürülen Değer:** `xTaskNotify()` aşağıda belirtilen durum dışında `pdPASS` döndürür.

Geçerli `xTaskNotify()` `eNotifyAction` Parametre Değerleri ve Bunların Alıcı Görevin Bildirim Değeri Üzerindeki Etkileri

- **`eNoAction`**: Alıcı görevin bildirim durumu (notification state), bildirim değeri güncellenmeden (without it's notification value being updated) beklemede (pending) olarak ayarlanır. `xTaskNotify()` `ulValue` parametresi kullanılmaz. `eNoAction` eylemi, görev bildiriminin ikili semafora (binary semaphore) daha hızlı ve daha hafif bir alternatif olarak kullanılmasına olanak tanır.
- **`eSetBits`**: Alıcı görevin bildirim değeri (notification value), `xTaskNotify()` `ulValue` parametresinde aktarılan değerle bitsel (bitwise) OR işlemine tabi tutulur. Örneğin, `ulValue` değeri `0x01` olarak ayarlanırsa, alıcı görevin bildirim değerinde bit 0 ayarlanır (set). Başka bir örnek olarak, `ulValue` `0x06` (ikili 0110) ise alıcı görevin bildirim değerinde bit 1 ve bit 2 ayarlanacaktır.
`eSetBits` eylemi, bir görev bildiriminin bir olay grubuna (event group) daha hızlı ve daha hafif bir alternatif olarak kullanılmasına olanak tanır.
- **`eIncrement`**: Alıcı görevin bildirim değeri artırılır (incremented).
`xTaskNotify()` `ulValue` parametresi kullanılmaz.
`eIncrement` eylemi, bir görev bildiriminin ikili (binary) veya sayma (counting) semaforuna daha hızlı ve daha hafif bir alternatif olarak kullanılmasına olanak tanır ve daha basit olan `xTaskNotifyGive()` API işlevine eşdeğerdir.
- **`eSetValueWithoutOverwrite`**: Alıcı görevin `xTaskNotify()` çağrılmadan önce bekleyen (pending) bir bildirimi varsa, hiçbir eylemde bulunulmaz (no action is taken) ve `xTaskNotify()` `pdFAIL` döndürür.
Eğer alıcı görevin `xTaskNotify()` çağrılmadan önce bekleyen bir bildirimi yoksa, alıcı görevin bildirim değeri `xTaskNotify()` `ulValue` parametresinde aktarılan değere (value) ayarlanır.
- **`eSetValueWithOverwrite`**: Alıcı görevin `xTaskNotify()` çağrılmadan önce bekleyen bir bildirimi olup olmadığına bakılmaksızın (regardless of), alıcı görevin bildirim değeri `xTaskNotify()` `ulValue` parametresinde aktarılan değere ayarlanır.

10.3.6 xTaskNotifyWait() API İşlevi

`xTaskNotifyWait()`, `ulTaskNotifyTake()` işlevinin daha yetenekli bir sürümüdür. İsteğe bağlı bir zaman aşımı (timeout) ile bir görevin, eğer henüz beklemede değilse, çağıran görevin bildirim durumunun beklemede (pending) olmasını beklemesine olanak tanır.

`xTaskNotifyWait()`, hem işleve girerken (on entry) hem de işlevden çıkarken (on exit) çağıran görevin bildirim değerindeki bitlerin temizlenmesi (cleared) için seçenekler sunar.

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,

                            uint32_t ulBitsToClearOnExit,

                            uint32_t *pulNotificationValue,

                            TickType_t xTicksToWait );
```

Liste 10.9 `xTaskNotifyWait()` API işlevi prototipi

`xTaskNotifyWait()` parametreleri ve dönüş değeri:

- **`ulBitsToClearOnEntry`:** Çağıran görevin `xTaskNotifyWait()` çağrısından önce bekleyen (pending) bir bildirimi yoksa, `ulBitsToClearOnEntry` içinde ayarlanan (set) tüm bitler işleve girildiğinde (on entry) görevin bildirim değerinde temizlenecektir (cleared).
  Örneğin, `ulBitsToClearOnEntry` değeri `0x01` ise, görevin bildirim değerinin 0. biti temizlenir.
  Başka bir örnek olarak, `ulBitsToClearOnEntry` değerini `0xffffffff` (ULONG_MAX) olarak ayarlamak görevin bildirim değerindeki tüm bitleri temizleyerek değeri etkili bir şekilde 0'a temizleyecektir.
- **`ulBitsToClearOnExit`:** Çağıran görev bir bildirim aldığı için veya `xTaskNotifyWait()` çağrıldığında halihazırda bekleyen bir bildirimi olduğu için `xTaskNotifyWait()` işlevinden çıkarsa (exits), `ulBitsToClearOnExit` içinde ayarlanan tüm bitler görev `xTaskNotifyWait()` işlevinden çıkmadan önce görevin bildirim değerinde temizlenecektir.
  Bitler, görevin bildirim değeri `*pulNotificationValue` (aşağıdaki `pulNotificationValue` açıklamasına bakın) içine kaydedildikten sonra temizlenir.
  Örneğin, `ulBitsToClearOnExit` değeri `0x03` ise, işlevden çıkmadan önce görevin bildirim değerinin 0. biti ve 1. biti temizlenir.
  `ulBitsToClearOnExit` değerini `0xffffffff` (ULONG_MAX) olarak ayarlamak, görevin bildirim değerindeki tüm bitleri temizleyerek değeri etkili bir şekilde 0'a temizleyecektir.
- **`pulNotificationValue`:** Görevin bildirim değerini dışarı aktarmak (pass out) için kullanılır. `*pulNotificationValue` değişkenine kopyalanan değer, `ulBitsToClearOnExit` ayarı nedeniyle herhangi bir bit temizlenmeden (cleared) önceki görev bildirim değeridir.
  `pulNotificationValue` isteğe bağlı bir parametredir ve gerekli değilse `NULL` olarak ayarlanabilir.
- **`xTicksToWait`:** Çağıran görevin (calling task) bildirim durumunun (notification state) beklemede (pending) olmasını beklemek için Engellenmiş (Blocked) durumunda kalması gereken maksimum süredir.
  Blok süresi tick periyotları (tick periods) cinsinden belirtilir, bu nedenle temsil ettiği mutlak zaman tick frekansına bağlıdır. `pdMS_TO_TICKS()` makrosu, milisaniye cinsinden belirtilen bir zamanı tick cinsinden belirtilen bir zamana dönüştürmek için kullanılabilir.
  `FreeRTOSConfig.h` dosyasında `INCLUDE_vTaskSuspend` 1 olarak ayarlanmış olması koşuluyla, `xTicksToWait` değerini `portMAX_DELAY` olarak ayarlamak görevin süresiz olarak (zaman aşımı olmadan) beklemesine neden olacaktır.
- **Döndürülen Değer:** İki olası dönüş değeri vardır:
  - `pdTRUE`: Bu, `xTaskNotifyWait()` işlevinin bir bildirim alındığı için veya `xTaskNotifyWait()` çağrıldığında çağıran görevin halihazırda (already) bekleyen bir bildirimi olduğu için döndüğünü gösterir.
    Bir blok süresi belirtilmişse (`xTicksToWait` sıfır değilse), çağıran görev bildirim durumunun beklemede olmasını beklemek üzere Engellenmiş (Blocked) durumuna yerleştirilmiş (placed), ancak blok süresi dolmadan önce bildirim durumu beklemede olarak ayarlanmış olabilir.

    - `pdFALSE`: Bu, çağıran görev bir görev bildirimi almadan `xTaskNotifyWait()` işlevinin döndüğünü gösterir.
      Eğer `xTicksToWait` sıfır değilse, çağıran görev bildirim durumunun beklemede (pending) olmasını beklemek üzere Engellenmiş durumunda tutulmuş (held), ancak belirtilen blok süresi bu gerçekleşmeden önce (before that happened) dolmuştur.

10.3.7 Çevresel Aygıt Sürücülerinde Kullanılan Görev Bildirimleri (Task Notifications Used in Peripheral Device Drivers): UART Örneği

Çevresel aygıt sürücüsü (peripheral driver) kütüphaneleri, donanım arayüzlerinde yaygın işlemleri gerçekleştiren işlevler sağlar. Bu tür kütüphanelerin sıklıkla sağlandığı çevresel aygıtlara örnek olarak Evrensel Asenkron Alıcılar ve Vericiler (UART'lar - Universal Asynchronous Receivers and Transmitters), Seri Çevresel Arayüz (SPI - Serial Peripheral Interface) bağlantı noktaları, analogdan dijitale dönüştürücüler (ADC'ler) ve Ethernet bağlantı noktaları verilebilir. Bu tür kütüphaneler tarafından tipik olarak sağlanan işlevlere örnek olarak bir çevresel birimi başlatma (initialize), bir çevresel birime veri gönderme ve bir çevresel birimden veri alma (receive) işlevleri verilebilir.

Çevresel birimler üzerindeki bazı işlemlerin tamamlanması nispeten uzun zaman alır. Yüksek hassasiyetli (high precision) bir ADC dönüşümü ve büyük bir veri paketinin UART üzerinden iletilmesi (transmission) bu tür işlemlere örnektir. Bu durumlarda sürücü kütüphanesi işlevi, işlemin ne zaman tamamlandığını belirlemek için çevresel birimin durum yazmaçlarını (status registers) yoklayacak (poll - repeatedly read) şekilde uygulanabilir. Bununla birlikte, bu şekilde yoklama yapmak (polling), üretken bir işlem yapılmadığı halde işlemci zamanının %100'ünü kullandığı için neredeyse her zaman israftır (wasteful). Atık (waste), çevresel birimi yoklayan bir görevin (polling a peripheral), üretken (productive) bir işlem yapması gereken daha düşük öncelikli bir görevin yürütülmesini engelleyebildiği çok görevli bir sistemde özellikle pahalıdır (expensive).

İsraf (wasted) edilen işlem süresi potansiyelini (potential) önlemek için, verimli (efficient) bir RTOS destekli (aware) aygıt sürücüsü kesme güdümlü (interrupt driven) olmalı ve uzun (lengthy) bir işlem başlatan bir göreve işlemin tamamlanması için Engellenmiş (Blocked) durumunda bekleme seçeneği vermelidir. Bu şekilde, uzun (lengthy) işlemi gerçekleştiren görev Engellenmiş durumundayken daha düşük öncelikli görevler çalışabilir ve hiçbir görev onu üretken (productively) bir şekilde kullanamadığı sürece işlem süresini kullanmaz.

RTOS kullanan (aware) sürücü kütüphanelerinin görevleri Engellenmiş durumuna (Blocked state) yerleştirmek için bir ikili semafor (binary semaphore) kullanması yaygın bir uygulamadır. Bu teknik (technique), bir UART portunda veri ileten RTOS uyumlu bir kütüphane işlevinin taslağını (outline) sağlayan Liste 10.10'da gösterilen sözde kod (pseudo code) ile gösterilmiştir. Liste 10.10'da:

- `xUART`, UART çevresel birimini (peripheral) tanımlayan ve durum bilgilerini tutan bir yapıdır (structure). Yapının (structure) `xTxSemaphore` üyesi

`SemaphoreHandle_t` tipinde bir değişkendir. Semaforun zaten oluşturulduğu varsayılmaktadır.

- `xUART_Send()` işlevi herhangi bir karşılıklı dışlama mantığı (mutual exclusion logic) içermez. Birden fazla görev `xUART_Send()` işlevini kullanacaksa, uygulama yazarının uygulamanın içindeki karşılıklı dışlamayı (mutual exclusion) yönetmesi gerekecektir. Örneğin, bir görevin `xUART_Send()` çağırmadan önce bir Mutex (mutex) elde etmesi (obtain) gerekebilir.
- `xSemaphoreTake()` API işlevi, UART iletimi başlatıldıktan (initiated) sonra çağıran görevi (calling task) Engellenmiş durumuna (Blocked state) yerleştirmek (place) için kullanılır.
- `xSemaphoreGiveFromISR()` API işlevi, UART çevre biriminin (peripheral) iletim sonu (transmit end) kesme hizmet rutini (interrupt service routine) yürütüldüğünde (executes), yani iletim tamamlandıktan sonra görevi (task) Engellenmiş durumundan çıkarmak için kullanılır.

```
/* UART'a veri göndermek için sürücü kütüphanesi işlevi. */
BaseType_t xUART_Send( xUART *pxUARTInstance,
                       uint8_t *pucDataSource,
                       size_t uxLength )
{
    BaseType_t xReturn;

    /* Zaman aşımı olmadan semaforu almayı deneyerek UART'ın
       iletim (transmit) semaforunun zaten mevcut olmadığından
       (not already available) emin olun. */
    xSemaphoreTake( pxUARTInstance->xTxSemaphore, 0 );

    /* İletimi (transmission) başlatın. */
    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* İletimin tamamlanmasını beklemek için semafor üzerinde engellenin.
       Eğer semafor alınırsa (obtained) xReturn değeri pdPASS olarak
       ayarlanacaktır. Eğer semafor alma (take) işlemi zaman aşımına
       uğrarsa (times out) xReturn değeri pdFAIL olarak ayarlanacaktır.
       UART_low_level_send() çağrısı ile xSemaphoreTake() çağrısı arasında
```

```
        kesme (interrupt) meydana gelirse (occurs), olayın (event) ikili semafora
        kilitleneceğine (latched) ve xSemaphoreTake() çağrısının hemen
        geri döneceğine (return immediately) dikkat edin. */
    xReturn = xSemaphoreTake( pxUARTInstance->xTxSemaphore,
                              pxUARTInstance->xTxTimeout );

    return xReturn;
}
/*-----------------------------------------------------------*/

/* Son bayt (last byte) UART'a gönderildikten sonra yürütülen UART'ın
   iletim sonu (transmit end) kesmesi (interrupt) için hizmet rutini. */
void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Kesmeyi (interrupt) temizleyin (Clear). */
    UART_low_level_interrupt_clear( pxUARTInstance );

    /* İletimin bittiğini bildirmek için Tx semaforunu verin (give). Eğer
       bir görev semaforu bekliyorsa (Blocked), görev Engellenmiş
       durumundan çıkarılacaktır. */
    xSemaphoreGiveFromISR( pxUARTInstance->xTxSemaphore,
                           &xHigherPriorityTaskWoken );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Liste 10.10 Bir sürücü kütüphanesi (driver library) iletim işlevinde (transmit function) ikili semaforun (binary semaphore) nasıl kullanılabileceğini gösteren sözde kod (pseudo code)

Liste 10.10'da gösterilen teknik mükemmel bir şekilde uygulanabilirdir (workable) ve gerçekten de (indeed) yaygın bir uygulamadır (common practice), ancak bazı dezavantajları (drawbacks) vardır:

- Kütüphane (library) birden fazla (multiple) semafor kullanır ve bu da RAM ayak izini (RAM footprint) artırır.
- Semaforlar oluşturulana (created) kadar kullanılamazlar, bu nedenle semafor kullanan bir kütüphane açıkça başlatılana (explicitly initialized) kadar kullanılamaz.
- Semaforlar, çok çeşitli (wide range of) kullanım durumlarına uygulanabilen (applicable) genel nesnelerdir (generic objects); herhangi bir sayıda görevin semaforun kullanılabilir olmasını beklemek (wait for the semaphore to become available) üzere Engellenmiş (Blocked) durumunda (state) beklemesine izin veren ve semafor kullanılabilir (available) olduğunda Engellenmiş durumundan hangi görevin (deterministik - deterministic bir şekilde) çıkarılacağını (remove) seçen mantığı (logic) içerirler. Bu mantığın (logic) yürütülmesi (Executing) sonlu (finite) bir zaman alır ve aynı anda semaforu (semaphore) bekleyen (waiting) birden fazla görev (task) olamayacağı Liste 10.10'da gösterilen senaryoda (scenario) bu işlem (processing) yükü (overhead) gereksizdir.

Liste 10.11, ikili bir semafor (binary semaphore) yerine bir görev bildirimi (task notification) kullanarak bu dezavantajlardan (drawbacks) nasıl kaçınılacağını (a`void`) göstermektedir.

Not: Eğer bir kütüphane görev bildirimleri kullanıyorsa, kütüphanenin dokümantasyonunda, bir kütüphane işlevinin çağrılmasının, çağıran görevin bildirim durumunu ve bildirim değerini değiştirebileceği açıkça (clearly) belirtilmelidir.

Liste 10.11'de:

- `xUART` yapısının (structure) `xTxSemaphore` üyesinin yerini `xTaskToNotify` üyesi almıştır. `xTaskToNotify`, `TaskHandle_t` türünde bir değişkendir (variable) ve UART işleminin (operation) tamamlanmasını bekleyen görevin tanıtıcısını (handle) tutmak için kullanılır.
- `xTaskGetCurrentTaskHandle()` FreeRTOS API işlevi, Çalışıyor (Running) durumundaki görevin tanıtıcısını (handle) elde etmek (obtain) için kullanılır.
- Kütüphane hiçbir FreeRTOS nesnesi (object) oluşturmaz, bu nedenle bir RAM yüküne (overhead) maruz kalmaz ve açıkça (explicitly) başlatılmasına (initialized) gerek yoktur.
- Görev bildirimi (task notification), doğrudan UART işleminin tamamlanmasını bekleyen göreve (task) gönderilir, bu nedenle gereksiz hiçbir mantık (logic) yürütülmez (executed).

`xUART` yapısının `xTaskToNotify` üyesine (member) hem bir görevden hem de bir kesme hizmet rutininden (interrupt service routine) erişilir, bu nedenle işlemcinin (processor) değerini (value) nasıl güncelleyeceğinin (update) dikkate alınması (consideration) gerekir:

- `xTaskToNotify` tek bir bellek yazma (memory write) işlemi (operation) ile güncelleniyorsa (updated), tıpkı Liste 10.11'de gösterildiği gibi kritik bir bölümün (critical section) dışında (outside) güncellenebilir. `xTaskToNotify` 32 bitlik bir değişkense (variable) (`TaskHandle_t` 32 bitlik bir türeyse) ve FreeRTOS'un çalıştığı işlemci 32 bitlik bir işlemciyse durum böyle (this would be the case) olacaktır.
- `xTaskToNotify` değerini güncellemek için birden fazla bellek yazma işlemi gerekiyorsa, `xTaskToNotify` yalnızca kritik bir bölüm (critical section) içinden (from within) güncellenmelidir — aksi takdirde kesme hizmet rutini `xTaskToNotify`'ye tutarsız (inconsistent) bir durumdayken (state) erişebilir. Bu durum (This would be the case), `xTaskToNotify` 32-bit bir değişken ise ve FreeRTOS'un üzerinde çalıştığı işlemci 16-bit bir işlemci ise, tüm 32-bit'leri (all 32-bits) güncellemek için iki adet 16-bit bellek yazma (memory write) işlemi gerekeceğinden geçerli (would be) olacaktır.

Dahili (Internally) olarak, FreeRTOS uygulaması (implementation) içinde (within), `TaskHandle_t` bir işaretçidir (pointer), bu nedenle `sizeof( TaskHandle_t )` her zaman `sizeof( void * )` değerine eşittir (equals).

```
/* UART'a veri göndermek için sürücü kütüphanesi işlevi. */

BaseType_t xUART_Send( xUART *pxUARTInstance,

                       uint8_t *pucDataSource,

                       size_t uxLength )

{

    BaseType_t xReturn;


    /* Bu işlevi çağıran görevin tanıtıcısını (handle) kaydedin.

       Kitap metni, aşağıdaki satırın kritik bir bölümle (critical section)

       korunması gerekip gerekmediğine dair notlar içerir. */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();


    /* ulTaskNotifyTake() işlevini xClearCountOnExit parametresi

       pdTRUE ve blok süresi 0 (engellemeyin) olarak ayarlayarak,

       çağıran görevin zaten bekleyen (pending) bir bildirimi

       olmadığından emin olun. */

    ulTaskNotifyTake( pdTRUE, 0 );
```

```
    /* İletimi (transmission) başlatın. */
    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* İletimin (transmission) tamamlandığı bildirilene (notified)
       kadar engellenin (Block). Bildirim alınırsa, ISR bu görevin
       bildirim değerini (notification value) 1'e (pdTRUE) çıkaracağından
       (increment) xReturn değeri 1 olarak ayarlanacaktır.
       İşlem (operation) zaman aşımına uğrarsa (times out), bu görevin bildirim
       değeri yukarıda 0'a temizlendiğinden (cleared) beri değiştirilmemiş
       olacağından xReturn 0 (pdFALSE) olacaktır. Kesme hizmet
       rutini (ISR) UART_low_level_send() çağrısı ile ulTaskNotifyTake()
       çağrısı arasında yürütülürse, olay (event) görevin bildirim
       değerine kilitlenecek (latched) ve ulTaskNotifyTake() çağrısı hemen
       (immediately) geri dönecektir (return). */
    xReturn = ( BaseType_t ) ulTaskNotifyTake( pdTRUE,
                                               pxUARTInstance->xTxTimeout );

    return xReturn;
}
/*-----------------------------------------------------------*/

/* Son bayt (last byte) UART'a gönderildikten sonra yürütülen ISR. */
void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Bildirilmeyi bekleyen bir görev (task waiting to be notified)
       olmadıkça (unless) bu işlev yürütülmemelidir. Bu durumu (condition) bir
assert ile test
       edin. Bu adım (step) kesinlikle (strictly) gerekli değildir (not
necessary),
```

```
        ancak hata ayıklamaya (debugging) yardımcı (aid) olacaktır.

        configASSERT() Bölüm 12.2'de açıklanmıştır. */

    configASSERT( pxUARTInstance->xTaskToNotify != NULL );


    /* Kesmeyi (interrupt) temizleyin (Clear). */

    UART_low_level_interrupt_clear( pxUARTInstance );


    /* Doğrudan xUART_Send() işlevini çağıran (called) göreve (task) bir

        bildirim gönderin. Görev bildirimi beklemek (waiting) üzere

        Engellenmiş ise (Blocked), görev Engellenmiş (Blocked) durumundan (state)
çıkarılacaktır (removed). */

    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,

                            &xHigherPriorityTaskWoken );


    /* Artık (Now) bildirilmeyi bekleyen bir görev yoktur. xUART

        yapısının (structure) xTaskToNotify üyesini (member) tekrar

        NULL olarak ayarlayın. Bu adım (step) kesinlikle gerekli değildir ancak

        hata ayıklamaya yardımcı olacaktır. */

    pxUARTInstance->xTaskToNotify = NULL;


    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Liste 10.11 Bir görev bildiriminin bir sürücü kütüphanesi (driver library) iletim işlevinde (transmit function) nasıl kullanılabileceğini gösteren sözde kod

Görev bildirimleri (Task notifications), Liste 10.12'de gösterilen ve bir UART bağlantı noktasından veri alan (receives) RTOS özellikli bir kütüphane işlevinin taslağını sağlayan sözde kodda (pseudo code) gösterildiği gibi, alma (receive) işlevlerinde semaforların (semaphores) yerini de (also) alabilir. Liste 10.12'ye referansla:

- `xUART_Receive()` işlevi herhangi bir karşılıklı dışlama mantığı (mutual exclusion logic) içermez (does not include). Birden fazla (more than one) görev `xUART_Receive()` işlevini kullanacaksa (going to use), uygulama yazarı (application writer) uygulamanın (application) kendisi içindeki (within)

karşılıklı dışlamayı yönetmek (manage) zorunda kalacaktır. Örneğin, bir görevin `xUART_Receive()` öğesini çağırmadan (calling) önce bir mutex alması gerekebilir.

- UART'ın alma (receive) kesme hizmet rutini (interrupt service routine), UART tarafından alınan (received) karakterleri (characters) bir RAM arabelleğine (RAM buffer) yerleştirir (places). `xUART_Receive()` işlevi RAM tamponundan (buffer) karakterleri döndürür (returns).
- `xUART_Receive()` `uxWantedBytes` parametresi alınacak (receive) karakter sayısını belirtmek (specify) için kullanılır. RAM arabelleği halihazırda istenen sayıda karakteri içermiyorsa, çağıran görev arabellekteki karakter sayısının arttığının (increased) bildirilmesini (notified) beklemek (wait) üzere Engellenmiş (Blocked) durumuna (state) yerleştirilir (placed). `while()` döngüsü (loop), alıcı (receive) arabelleği istenen (requested) sayıda karakteri içerene (contains) veya bir zaman aşımı (timeout) meydana gelene kadar (until) bu diziyi (sequence) tekrarlamak (repeat) için kullanılır.
- Çağıran görev Engellenmiş (Blocked) durumuna (state) birden fazla kez girebilir. Bu nedenle blok süresi (block time), `xUART_Receive()` çağrıldığından (called) beri geçen süreyi (amount of time) hesaba katacak şekilde ayarlanır (adjusted). Ayarlamalar (adjustments), `xUART_Receive()` içinde geçirilen (spent) toplam sürenin (total time) `xUART` yapısının `xRxTimeout` üyesi (member) tarafından belirtilen blok süresini (block time) aşmamasını (not exceed) sağlar. Blok süresi FreeRTOS `vTaskSetTimeOutState()` ve `xTaskCheckForTimeOut()` yardımcı işlevleri (helper functions) kullanılarak ayarlanır.

```
/* Bir UART'tan veri almak (receive) için sürücü kütüphanesi (Driver library)
işlevi. */

size_t xUART_Receive( xUART *pxUARTInstance,

                      uint8_t *pucBuffer,

                      size_t uxWantedBytes )

{

    size_t uxReceived = 0;

    TickType_t xTicksToWait;

    TimeOut_t xTimeOut;



    /* Bu işleve (function) girildiği zamanı (time) kaydedin (Record). */

    vTaskSetTimeOutState( &xTimeOut );



    /* xTicksToWait zaman aşımı (timeout) değeridir - başlangıçta bu UART

       örneği (instance) için maksimum alma (receive) zaman aşımına
```

```
       ayarlanmıştır. */

    xTicksToWait = pxUARTInstance->xRxTimeout;


    /* Bu işlevi çağıran görevin (task) tanıtıcısını (handle) kaydedin (Save).

       Kitap metni, aşağıdaki satırın kritik bir bölümle (critical section)

       korunması gerekip gerekmediğine dair (whether) notlar (notes) içerir
(contains). */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();


    /* Arabellek (buffer) istenen (wanted) sayıda bayt (bytes) içerene

       veya (or) bir zaman aşımı (timeout) meydana gelene (occurs) kadar

       döngüye girin (Loop). */

    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )

    {

        /* Şimdiye kadar bu işlevde harcanan zamanı (time spent) hesaba katmak

           (account for) için xTicksToWait değerini ayarlayarak (adjusting) bir

           zaman aşımı (timeout) olup olmadığına (look for) bakın. */

        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )

        {

            /* İstenen (wanted) sayıda bayt (bytes) kullanıma (available)

               sunulmadan (before) önce (timed out) zaman aşımına uğradı (Timed
out),

               döngüden çıkın (exit). */

            break;

        }


        /* Alma (receive) arabelleği (buffer) henüz istenen miktarda

           (required amount of) bayt (bytes) içermemektedir (does not yet
contain).

           Alma (receive) kesme hizmet rutininin (interrupt service routine)
arabelleğe (buffer)
```

```
            daha fazla veri (more data) yerleştirdiğinin (placed) bildirilmesi
(notified)

            için en fazla (maximum of) xTicksToWait tick bekleyin (Wait for).
Çağıran görevin (calling task)

            bu işlevi (function) çağırdığında (called) zaten (already) bekleyen
(pending)

            bir bildirimi (notification) olması önemli değildir (does not matter),
eğer

            öyleyse (if it did), bu while döngüsü (loop) etrafında

            fazladan (extra) bir kez daha yinelenecektir (iterate). */

        ulTaskNotifyTake( pdTRUE, xTicksToWait );

    }



    /* Bildirimleri almak (receive) için bekleyen (waiting) hiçbir görev (no
tasks)

       yoktur, bu nedenle (so) xTaskToNotify öğesini tekrar (back to) NULL

       olarak ayarlayın. Kitap metni, aşağıdaki satırın kritik bir bölümle

       (critical section) korunması gerekip gerekmediğine dair notlar içerir. */

    pxUARTInstance->xTaskToNotify = NULL;



    /* Alıcı (receive) arabelleğinden (buffer) uxWantedBytes okuyarak

       pucBuffer içine (into) yazmayı (Attempt to read) deneyin.

       Okunan gerçek bayt sayısı (uxWantedBytes'tan daha az

       (less than) olabilir) döndürülür (returned). */

    uxReceived = UART_read_from_receive_buffer( pxUARTInstance,

                                                pucBuffer,

                                                uxWantedBytes );



    return uxReceived;

}

/*-----------------------------------------------------------*/
```

```
/* UART'ın alma (receive) kesmesi (interrupt) için kesme hizmet rutini (interrupt
service routine) */

void xUART_ReceiveISR( xUART *pxUARTInstance )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;



    /* Alınan (received) verileri (data) bu UART'ın alma (receive) arabelleğine

       (buffer) kopyalayın ve kesmeyi (interrupt) temizleyin. */

    UART_low_level_receive( pxUARTInstance );



    /* Eğer bir görev yeni veri bildirimini (notified of the new data) bekliyorsa

       şimdi ona bildirin. */

    if( pxUARTInstance->xTaskToNotify != NULL )

    {

        vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,

                                &xHigherPriorityTaskWoken );

        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

    }

}
```

Liste 10.12 Bir sürücü kütüphanesi (driver library) alma işlevinde (receive function) bir görev bildiriminin (task notification) nasıl kullanılabileceğini gösteren sözde kod (Pseudo code)

10.3.8 Çevresel Aygıt Sürücülerinde Kullanılan Görev Bildirimleri (Task Notifications Used in Peripheral Device Drivers): ADC Örneği

Önceki bölüm, bir kesmeden (interrupt) bir göreve görev bildirimi göndermek için `vTaskNotifyGiveFromISR()` işlevinin nasıl kullanılacağını göstermiştir. `vTaskNotifyGiveFromISR()` kullanımı basit (simple) bir işlevdir, ancak yetenekleri (capabilities) sınırlıdır; yalnızca değersiz (valueless) bir olay olarak bir görev bildirimi gönderebilir, veri gönderemez. Bu bölümde, bir görev bildirimi (task notification) olayıyla veri göndermek için `xTaskNotifyFromISR()` işlevinin nasıl kullanılacağı gösterilmektedir.

Bu teknik (technique), Liste 10.13'te gösterilen ve bir Analog-Dijital Dönüştürücü (ADC) için RTOS'a duyarlı (aware) bir kesme hizmet rutininin (interrupt service routine) taslağını sağlayan sözde kodla gösterilmektedir. Liste 10.13'te:

- En az (at least) her 50 milisaniyede bir ADC dönüşümünün başlatıldığı varsayılmaktadır.
- `ADC_ConversionEndISR()`, her yeni ADC değeri kullanılabilir (available) olduğunda yürütülen (executes) kesme olan ADC'nin dönüştürme sonu (conversion end) kesmesi için kesme hizmet rutinidir (interrupt service routine).
- `vADCTask()` tarafından uygulanan görev, ADC tarafından üretilen (generated) her değeri işler. Görevin tanıtıcısının görev oluşturulduğunda `xADCTaskToNotify` içinde saklandığı varsayılmaktadır.
- `ADC_ConversionEndISR()`, `vADCTask()` görevine bir görev bildirimi göndermek ve ADC dönüştürmesinin (conversion) sonucunu görevin bildirim değerine (notification value) yazmak için `eAction` parametresini `eSetValueWithoutOverwrite` olarak ayarlayarak `xTaskNotifyFromISR()` işlevini kullanır.
- `vADCTask()` görevi, yeni bir ADC değerinin mevcut olduğunun bildirilmesini beklemek (wait to be notified) ve ADC dönüştürmesinin (conversion) sonucunu bildirim değerinden (notification value) almak (retrieve) için `xTaskNotifyWait()` işlevini kullanır.

```
/* ADC kullanan (uses) bir görev. */

void vADCTask( void *pvParameters )

{

    uint32_t ulADCValue;

    BaseType_t xResult;


    /* ADC dönüşümlerinin (conversions) tetiklenme (triggered) oranı (rate). */

    const TickType_t xADCConversionFrequency = pdMS_TO_TICKS( 50 );


    for( ;; )

    {

        /* Bir sonraki ADC dönüşüm (conversion) sonucunu (result) bekleyin. */

        xResult = xTaskNotifyWait(

            /* Yeni ADC değeri eski değerin üzerine yazılacaktır (overwrite), bu

               nedenle yeni bildirim (notification) değerini beklemeden
```

```
                  önce herhangi bir biti temizlemeye (clear any bits)
                  gerek (no need) yoktur. */
               0,
               /* Gelecekteki ADC değerleri mevcut değerin (existing value) üzerine
                  yazılacaktır, bu nedenle xTaskNotifyWait() öğesinden çıkmadan
                  (exiting) önce herhangi bir biti temizlemeye (clear any bits)
                  gerek yoktur. */
               0,
               /* Görevin bildirim değerinin (en son - latest ADC dönüştürme
                  sonucunu tutar) kopyalanacağı (will be copied) değişkenin
                  (variable) adresi (address). */
               &ulADCValue,
               /* Her xADCConversionFrequency tick süresinde bir
                  yeni bir ADC değeri (value) alınmalıdır. */
               xADCConversionFrequency * 2 );

        if( xResult == pdPASS )
        {
            /* Yeni bir ADC değeri alındı. Şimdi (now) onu işleyin. */
            ProcessADCResult( ulADCValue );
        }
        else
        {
            /* xTaskNotifyWait() çağrısı beklenen (expected) süre içinde (within)
               dönmedi, ADC dönüşümünü (conversion) tetikleyen (triggers)
               girişte (input) veya ADC'nin kendisinde bir sorun
               (something must be wrong) olmalı. Hatayı (error) burada işleyin. */
        }
    }
}
```

```
/*-----------------------------------------------------------*/


/* Her bir ADC dönüşümü (conversion) tamamlandığında yürütülen (executes)

   kesme hizmet rutini (interrupt service routine). */

void ADC_ConversionEndISR( xADC *pxADCInstance )

{

    uint32_t ulConversionResult;

    BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;


    /* Yeni ADC değerini okuyun (read) ve kesmeyi (interrupt) temizleyin. */

    ulConversionResult = ADC_low_level_read( pxADCInstance );


    /* Bir bildirimi (notification) ve ADC dönüşüm (conversion) sonucunu (result)
doğrudan

       vADCTask() işlevine gönderin. */

    xResult = xTaskNotifyFromISR( xADCTaskToNotify,          /* xTaskToNotify
parametresi */

                                  ulConversionResult,        /* ulValue
parametresi */

                                  eSetValueWithoutOverwrite, /* eAction
parametresi. */

                                  &xHigherPriorityTaskWoken );


    /* Eğer xTaskNotifyFromISR() çağrısı pdFAIL döndürürse (returns), görev

       ADC değerlerinin (values) üretilme (generated) hızına (rate) ayak

       uyduramıyordur (not keeping up).

       configASSERT() Bölüm 11.2'de açıklanmıştır. */

    configASSERT( xResult == pdPASS );


    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Liste 10.13 Bir görev bildiriminin bir göreve bir değer (value) aktarmak (pass) için nasıl kullanılabileceğini gösteren sözde kod (pseudo code)

10.3.9 Bir Uygulama İçinde Doğrudan (Directly) Kullanılan Görev Bildirimleri (Task Notifications)

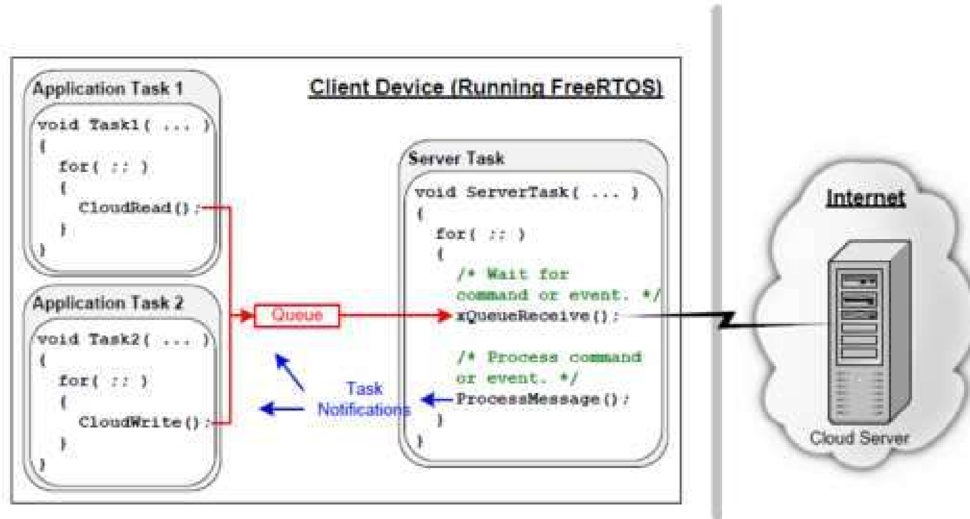
Bu bölüm, görev bildirimlerinin gücünü, onları aşağıdaki (following) işlevselliği içeren varsayımsal (hypothetical) bir uygulamada kullanımlarını (use) göstererek (demonstrating) pekiştirmektedir (reinforces):

- Uygulama, uzak bir veri sunucusuna (remote data server) veri göndermek ve ondan veri istemek (request) için yavaş (slow) bir internet bağlantısı (connection) üzerinden (across) iletişim kurar. Buradan itibaren (From here on), uzak veri sunucusuna bulut sunucusu (cloud server) olarak atıfta bulunulmaktadır (referred to as).
- Bulut sunucusundan veri istedikten sonra, istekte bulunan görev (requesting task), istenen (requested) verilerin alınması (received) için Engellenmiş durumunda beklemelidir (must wait).
- Bulut sunucusuna veri gönderdikten sonra (after sending), gönderen görev (sending task), bulut sunucusunun verileri doğru bir şekilde (correctly) aldığına dair bir onay (acknowledgement) için Engellenmiş durumunda beklemelidir.

Yazılım tasarımının (software design) bir şeması (schematic) Şekil 10.6'da gösterilmiştir. Şekil 10.6'da:

- Bulut sunucusuna birden fazla (multiple) internet bağlantısını (connections) idare etmenin (handling) karmaşıklığı (complexity) tek bir (single) FreeRTOS görevi (task) içinde (within) kapsüllenmiştir (encapsulated). Görev, FreeRTOS uygulaması (application) içinde bir proxy sunucusu (proxy server) görevi görür (acts as) ve sunucu görevi (server task) olarak adlandırılır.
- Uygulama görevleri `CloudRead()` işlevini çağırarak (calling) bulut sunucusundan veri okur (read). `CloudRead()` bulut sunucusu ile doğrudan iletişim kurmaz, bunun yerine (instead) okuma isteğini (read request) bir kuyruk (queue) üzerinde (on) sunucu görevine gönderir (sends) ve istenen (requested) veriyi sunucu görevinden bir görev bildirimi (task notification) olarak alır.
- Uygulama görevleri `CloudWrite()` öğesini çağırarak bulut sunucusuna (cloud server) veri yazar (write). `CloudWrite()` bulut sunucusu ile doğrudan iletişim kurmaz (communicate), bunun yerine (instead) yazma isteğini (write request) bir kuyruk (queue) üzerinden sunucu görevine gönderir ve yazma işleminin (operation) sonucunu sunucu görevinden bir görev bildirimi (task notification) olarak alır.

`CloudRead()` ve `CloudWrite()` işlevleri tarafından sunucu görevine (server task) gönderilen yapı (structure) Liste 10.14'te gösterilmektedir.



Şekil 10.6 Uygulama görevlerinden bulut sunucusuna ve bulut sunucusundan uygulama görevlerine iletişim yolları

```
typedef enum CloudOperations
{
    eRead,  /* Bulut sunucusundan veri almak için. */
    eWrite  /* Bulut sunucusuna veri göndermek için. */
} Operation_t;


typedef struct CloudCommand
{
    Operation_t eOperation; /* Gerçekleştirilecek işlem (okuma veya yazma). */
    uint32_t ulDataID;      /* Okunan veya yazılan veriyi tanımlar. */
    uint32_t ulDataValue;   /* Sadece bulut sunucusuna veri yazarken kullanılır.
*/
    TaskHandle_t xTaskToNotify; /* İşlemi gerçekleştiren görevin tanıtıcısı. */
} CloudCommand_t;
```

Liste 10.14 Sunucu görevine bir kuyruk (queue) üzerinde gönderilen (sent) yapı (structure) ve veri türü (data type)

`CloudRead()` için sözde kod (pseudo code) Liste 10.15'te gösterilmektedir. İşlev isteğini (request) sunucu görevine (server task) gönderir, ardından istenen (requested)

verilerin kullanıma (available) sunulduğu (notified) bildirilene kadar Engellenmiş durumunda beklemek için `xTaskNotifyWait()` işlevini çağırır.

Sunucu görevinin (server task) bir okuma isteğini (read request) nasıl yönettiğini (manages) gösteren sözde kod Liste 10.16'da gösterilmektedir. Veriler bulut sunucusundan alındığında, sunucu görevi uygulama görevinin blokajını (unblocks) kaldırır ve `eAction` parametresi `eSetValueWithOverwrite` olarak ayarlanmış olarak `xTaskNotify()` işlevini çağırarak (calling) alınan (received) verileri uygulama görevine gönderir (sends).

Liste 10.16, `GetCloudData()` işlevinin bulut sunucusundan bir değer (value) elde etmek (obtain) için beklemesi (wait) gerekmediğini varsaydığı (assumes) için basitleştirilmiş (simplified) bir senaryoyu göstermektedir.

```
/* ulDataID okunacak veriyi (data to read) tanımlar. pulValue bulut

   sunucusundan alınan verinin yazılacağı değişkenin adresini (address

   of the variable) tutar. */

BaseType_t CloudRead( uint32_t ulDataID, uint32_t *pulValue )

{

    CloudCommand_t xRequest;

    BaseType_t xReturn;


    /* CloudCommand_t yapı üyelerini (structure members) bu okuma

       isteği (read request) için doğru (correct) olacak şekilde ayarlayın. */

    xRequest.eOperation = eRead; /* Bu bir veri okuma isteğidir (request). */

    xRequest.ulDataID = ulDataID; /* Okunacak veriyi tanımlayan (identifies) bir
kod. */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Çağıran görevin
(calling task)

                                                              tanıtıcısı. */


    /* Bildirim değerini (notification value) 0'lık bir blok süresiyle (block
time)

       okuyarak halihazırda bekleyen (already pending) hiçbir bildirim
olmadığından (ensure)

       emin olun, ardından (then) yapıyı (structure) sunucu görevine gönderin. */

    xTaskNotifyWait( 0, 0, NULL, 0 );
```

```
    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );



    /* Sunucu görevinden (server task) bir bildirim (notification) bekleyin.

       Sunucu görevi (server task), bulut sunucusundan aldığı değeri (value)
doğrudan (directly)

       bu görevin bildirim değerine yazar, bu nedenle (so there is no need)

       xTaskNotifyWait() işlevine girerken (on entry) veya çıkarken (on exit)
bildirim değerindeki

       herhangi bir biti temizlemeye (clear any bits) gerek yoktur. Alınan değer
*pulValue'ya

       yazılır, bu nedenle pulValue bildirim değerinin (notification value)
yazıldığı

       adres olarak geçirilir (passed as). */

    xReturn = xTaskNotifyWait( 0,          /* Girişte temizlenen bit yok (No bits
cleared on entry) */

                               0,          /* Çıkışta temizlenecek bit yok (No
bits to clear on exit) */

                               pulValue,   /* Bildirim değeri *pulValue içine
yazılır */

                               pdMS_TO_TICKS( 250 ) ); /* Maksimum 250 ms bekleyin
*/



    /* Eğer xReturn pdPASS ise, değer elde edilmiştir. Eğer

       xReturn pdFAIL ise, istek (request) zaman aşımına uğramıştır (timed out).
*/

    return xReturn;

}
```

Liste 10.15 Bulut Okuma (Cloud Read) API İşlevinin Uygulaması (Implementation)

```
void ServerTask( void *pvParameters )

{

    CloudCommand_t xCommand;

    uint32_t ulReceivedValue;


```

```
    for( ;; )

    {

        /* Bir görevden alınacak (received) bir sonraki (next) CloudCommand_t
yapısını bekleyin */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );



        switch( xCommand.eOperation ) /* Okuma mı yoksa yazma isteği miydi? */

        {

            case eRead:

                /* Uzak (remote) bulut sunucusundan istenen (requested) veri
öğesini (data item) elde edin */

                ulReceivedValue = GetCloudData( xCommand.ulDataID );



                /* İsteği (request) yapan (made) göreve hem (both) bir bildirim
hem de (and)

                   bulut sunucusundan alınan (received) değeri göndermek (send)
için xTaskNotify()

                   işlevini çağırın. Görevin tanıtıcısı (handle) CloudCommand_t

                   yapısından elde edilir. */

                xTaskNotify( xCommand.xTaskToNotify, /* Görevin tanıtıcısı yapının
içindedir */

                             ulReceivedValue,        /* Bildirim değeri olarak
gönderilen bulut verisi */

                             eSetValueWithOverwrite );

                break;



            /* Diğer (other) switch durumları (cases) buraya gelir. */

        }

    }

}
```

Liste 10.16 Okuma İsteğini (Read Request) İşleyen (Processing) Sunucu Görevi

`CloudWrite()` için sözde kod Liste 10.17'de gösterilmektedir. Gösterim (demonstration) amacıyla `CloudWrite()`, durum kodundaki her bite benzersiz (unique) bir anlam atandığı bit tabanlı (bitwise) bir durum (status) kodu döndürür. Dört örnek durum biti (status bits), Liste 10.17'nin en üstündeki (top) `#define` ifadeleriyle (statements) gösterilmektedir.

Görev dört durum bitini temizler (clears), isteğini sunucu görevine gönderir, ardından durum bildirimini (status notification) Engellenmiş durumunda beklemek için `xTaskNotifyWait()` işlevini çağırır.

```
/* Bulut yazma (cloud write) işlemi (operation) tarafından kullanılan durum
bitleri (Status bits). */

#define SEND_SUCCESSFUL_BIT             ( 0x01 << 0 )

#define OPERATION_TIMED_OUT_BIT         ( 0x01 << 1 )

#define NO_INTERNET_CONNECTION_BIT      ( 0x01 << 2 )

#define CANNOT_LOCATE_CLOUD_SERVER_BIT  ( 0x01 << 3 )



/* Dört durum bitinin de (four status bits) ayarlandığı (set) bir maske (mask). */

#define CLOUD_WRITE_STATUS_BIT_MASK     ( SEND_SUCCESSFUL_BIT | \

                                          OPERATION_TIMED_OUT_BIT | \

                                          NO_INTERNET_CONNECTION_BIT | \

                                          CANNOT_LOCATE_CLOUD_SERVER_BIT )



uint32_t CloudWrite( uint32_t ulDataID, uint32_t ulDataValue )

{

    CloudCommand_t xRequest;

    uint32_t ulNotificationValue;



    /* CloudCommand_t yapısı (structure) üyelerini (members) bu

       yazma (write) isteği için doğru olacak şekilde ayarlayın. */

    xRequest.eOperation = eWrite; /* Bu bir veri yazma isteğidir */

    xRequest.ulDataID = ulDataID; /* Yazılan veriyi tanımlayan (identifies) bir
kod */

    xRequest.ulDataValue = ulDataValue; /* Bulut sunucusuna yazılan (written)
verinin (data)
```

```
                                        değeri (value). */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Çağıran (calling)
görevin

                                                            tanıtıcısı. */



    /* ulBitsToClearOnExit parametresini CLOUD_WRITE_STATUS_BIT_MASK olarak

       ve blok süresini (block time) 0 olarak ayarlayarak xTaskNotifyWait()

       işlevini çağırıp yazma işlemiyle ilgili (relevant to) üç durum

       bitini (three status bits) temizleyin (Clear). Mevcut bildirim (current
notification)

       değeri gerekli değildir, bu nedenle (so) pulNotificationValue

       parametresi NULL olarak ayarlanmıştır. */

    xTaskNotifyWait( 0, CLOUD_WRITE_STATUS_BIT_MASK, NULL, 0 );



    /* İsteği sunucu görevine (server task) gönderin. */

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );



    /* Sunucu görevinden (server task) bir bildirim bekleyin. Sunucu görevi
(server task),

       ulNotificationValue değişkenine yazılan (written to) bit tabanlı (bitwise)

       bir durum kodunu (status code) bu görevin bildirim (notification) değerine
yazar. */

    xTaskNotifyWait( 0,                           /* Girişte (on entry) temizlenen
bit yok. */

                     CLOUD_WRITE_STATUS_BIT_MASK, /* Çıkışta (on exit) ilgili
(relevant) bitleri 0'a temizleyin. */

                     &ulNotificationValue,        /* Bildirilen değer (Notified
value). */

                     pdMS_TO_TICKS( 250 ) );      /* Maksimum 250 ms bekleyin. */



    /* Durum kodunu çağıran göreve döndürün (return). */

    return ( ulNotificationValue & CLOUD_WRITE_STATUS_BIT_MASK );

}
```

Liste 10.17 Bulut Yazma (Cloud Write) API İşlevinin Uygulaması (Implementation)

Sunucu görevinin bir yazma isteğini (write request) nasıl yönettiğini (manages) gösteren sözde kod (pseudo code) Liste 10.18'de gösterilmektedir. Veriler bulut sunucusuna (cloud server) gönderildiğinde, sunucu görevi (server task) uygulama görevinin (application task) blokajını kaldırır (unblocks) ve `eAction` parametresi `eSetBits` olarak ayarlanmış şekilde `xTaskNotify()` işlevini çağırarak bit tabanlı durum kodunu (bitwise status code) uygulama görevine gönderir. Alıcı görevin bildirim değerinde yalnızca `CLOUD_WRITE_STATUS_BIT_MASK` sabiti (constant) tarafından tanımlanan bitler (bits) değiştirilebilir (altered), bu nedenle alıcı görev bildirim değerindeki diğer bitleri (other bits) başka amaçlar (purposes) için kullanabilir.

Liste 10.18, `SetCloudData()` işlevinin uzak bulut sunucusundan (remote cloud server) bir onay (acknowledgement) almak (obtain) için beklemesi (wait) gerekmediğini varsaydığı (assumes) için basitleştirilmiş bir senaryo göstermektedir.

```
void ServerTask( void *pvParameters )

{

    CloudCommand_t xCommand;

    uint32_t ulBitwiseStatusCode;


    for( ;; )

    {

        /* Bir sonraki (next) mesajı bekleyin. */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );


        /* Okuma (read) mı yoksa yazma (write) isteği (request) miydi? */

        switch( xCommand.eOperation )

        {

            case eWrite:

                /* Verileri uzak bulut sunucusuna gönderin. SetCloudData(),

                   yalnızca CLOUD_WRITE_STATUS_BIT_MASK tanımıyla (definition)

                   (Liste 10.17'de gösterilmiştir) tanımlanan bitleri kullanan

                   bit düzeyinde (bitwise) bir durum kodu (status code) döndürür.
*/

                ulBitwiseStatusCode = SetCloudData( xCommand.ulDataID,
```

```
                                                      xCommand.ulDataValue );


                /* Yazma (write) isteğini yapan göreve (task that made the
request)
                   bir bildirim (notification) gönderin. eSetBits eylemi (action)
kullanılır,
                   böylece ulBitwiseStatusCode içinde ayarlanan herhangi bir durum
                   biti bildirilen görevin bildirim değerinde (notification value)
                   ayarlanacaktır. Diğer (other) tüm bitler değişmeden (unchanged)
kalır.
                   Görevin tanıtıcısı CloudCommand_t yapısından (structure)
                   elde edilir (obtained). */
                xTaskNotify( xCommand.xTaskToNotify, /* Görevin tanıtıcısı yapının
içindedir. */
                             ulBitwiseStatusCode,    /* Bildirim değeri olarak
gönderilen bulut verisi. */
                             eSetBits );
                break;


            /* Diğer switch durumları (cases) buraya gelir. */
        }
    }
}
```

Liste 10.18 Bir Gönderme İsteğini (Send Request) İşleyen Sunucu Görevi (Server Task)

Bölüm 11

Düşük Güç Desteği (Low Power Support)

11.1 Güç Tasarrufuna Giriş (Power Saving Introduction)

FreeRTOS, IDLE görev kancaları (task hooks) ve tickless (kene/zaman işareti olmayan) Boşta (Idle) modu ile düşük güç (low power) modlarına geçmek (tap into) için kolay bir yol (easy way) sunar.

FreeRTOS'un çalıştığı mikrodenetleyici tarafından tüketilen (consumed) gücü (power), mikrodenetleyiciyi düşük güç durumuna (low power state) yerleştirmek için IDLE (Boşta) görev kancasını kullanarak azaltmak yaygındır (common). Bu yöntemle (method) elde edilebilecek (achieved) güç tasarrufu (power saving), tick kesmelerini (tick interrupts) işlemek için periyodik olarak düşük güç durumundan çıkma (exit) ve ardından tekrar girme (re-enter) zorunluluğu (necessity) ile sınırlıdır (limited). Ayrıca, tick kesmesinin (tick interrupt) frekansı (frequency) çok yüksekse (boşta durumundan - idle - uyanma - wake - çok sıksa), her tick için düşük bir güç durumuna girmek ve ardından çıkmak için tüketilen (consumed) enerji (energy) ve zaman (time), en hafif (lightest) güç tasarrufu modları dışındaki (all but) tüm modlar için potansiyel (potential) güç tasarrufu kazanımlarından (gains) daha ağır basacaktır (outweigh).

FreeRTOS, mikrodenetleyicinin (microcontroller) periyodik olarak (periodically) düşük güç tüketimine girmesine ve çıkmasına (enter and exit) izin veren düşük bir güç durumunu destekler. FreeRTOS tickless boşta (idle) modu, boşta (idle) kalma süreleri (periods) boyunca (yürütülebilecek hiçbir uygulama görevi olmadığında) periyodik tick kesmesini durdurur, bu da bir kesme (interrupt) meydana gelene (occurs) veya RTOS çekirdeğinin (kernel) bir görevi Hazır (Ready) durumuna (state) geçirme (transition) zamanı gelene kadar MCU'nun derin bir güç tasarrufu durumunda (deep power saving state) kalmasına (remain) olanak tanır. Daha sonra (then), tick kesmesi (tick interrupt) yeniden başlatıldığında (restarted) RTOS tick sayımı (count) değerinde düzeltici (correcting) bir ayarlama (adjustment) yapar. FreeRTOS tickless modunun (tickless mode) prensibi (principle), MCU boşta görevini (idle task) gerçekleştirirken (performing) sistem güç tüketiminden tasarruf etmek (save) için MCU'nun düşük güç moduna girmesini sağlamaktır.

11.2 FreeRTOS Uyku Modları (Sleep Modes)

FreeRTOS'ta desteklenen (supported) üç (three) tür (types) uyku modu (sleep modes) vardır:

1. **eAbortSleep** - Bu mod (mode), bir görevin hazır (ready) hale getirildiğini (made ready), bir bağlam anahtarının (context switch) beklemede (pended) olduğunu veya bir tick kesmesinin halihazırda (already) meydana geldiğini ancak çizgeleyici (scheduler) askıya alındığından (suspended) beri beklemede olduğunu (pended) belirtir (denotes). RTOS'a bir uyku moduna (sleep mode) girmeyi iptal etmesini (abort) bildirir (signals).
2. **eStandardSleep** - Bu mod, beklenen (expected) boşta kalma (idle) süresinden daha uzun (longer) sürmeyecek (will not last) bir uyku moduna girmeye izin verir (allows).
3. **`eNoTasksWaitingTimeout`** - Hiçbir görev bir zaman aşımı (timeout) beklemediğinde (no tasks are waiting) bu moda girilir (entered), bu nedenle yalnızca harici (external) bir kesme (interrupt) veya sıfırlama (reset) ile çıkılabilen (exited) bir uyku moduna girmek güvenlidir (safe).

11.3 İşlevler (Functions) ve Yerleşik (Built-in) Tickless Boşta (Idle) İşlevselliğini (Functionality) Etkinleştirme

Yerleşik (Built-in) Tickless Boşta işlevselliği, `FreeRTOSConfig.h` dosyasında `configUSE_TICKLESS_IDLE` 1 olarak tanımlanarak (defining) etkinleştirilir (enabled) (bu özelliği - feature - destekleyen bağlantı noktaları - ports - için). Kullanıcı (User) tanımlı (defined) tickless boşta (idle) işlevselliği, `FreeRTOSConfig.h` dosyasında `configUSE_TICKLESS_IDLE` değeri 2 olarak tanımlanarak (defining) herhangi bir FreeRTOS bağlantı noktası (yerleşik - built in - bir uygulama - implementation - içerenler - include - dahil) için sağlanabilir (provided).

Tickless boşta (idle) işlevselliği (functionality) etkinleştirildiğinde (enabled), aşağıdaki (following) iki koşul (conditions) karşılandığında (satisfied) çekirdek (kernel) `portSUPPRESS_TICKS_AND_SLEEP()` makrosunu çağıracaktır (call):

1. Tüm uygulama (application) görevleri Engellenmiş (Blocked) durumunda veya Askıya Alınmış (Suspended) durumunda (state) olduğundan, Boşta (Idle) görevi çalışabilen (able to run) tek (only) görevdir (task).
2. Çekirdeğin (kernel) bir uygulama (application) görevini Engellenmiş durumundan çıkarmasından (transition out) önce (before) en az (at least) `n` kadar (further) tam (complete) tick periyodu (periods) geçecektir (pass), burada `n`, `FreeRTOSConfig.h` içindeki `configEXPECTED_IDLE_TIME_BEFORE_SLEEP` tanımı (definition) ile ayarlanır (set).

11.3.1 portSUPPRESS_TICKS_AND_SLEEP() Makrosu

```
portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime )
```

Liste 11.1 `portSUPPRESS_TICKS_AND_SLEEP` makrosu için prototip (prototype)

`portSUPPRESS_TICKS_AND_SLEEP()` içindeki `xExpectedIdleTime` parametresinin (parameter) değeri (value), bir görevin Hazır (Ready) durumuna (state) geçirilmesinden (moved) önceki (before) toplam (total) tick periyodu (periods) sayısına eşittir (equals). Bu nedenle parametre değeri, mikrodenetleyicinin (microcontroller) tick kesmesi (tick interrupt) bastırılmış (suppressed) halde derin bir uyku (deep sleep) durumunda güvenle (safely) kalabileceği (remain) süredir (time).

11.3.2 vPortSuppressTicksAndSleep İşlevi

`vPortSuppressTicksAndSleep()` işlevi (function) FreeRTOS'ta tanımlanmıştır (defined) ve tickless modunu (mode) uygulamak (implement) için kullanılabilir. Bu işlev, FreeRTOS Cortex-M bağlantı noktası (port) katmanında (layer) zayıf (weakly) bir şekilde tanımlanmıştır ve uygulama yazarı (application writer) tarafından geçersiz kılınabilir (overridden).

```
void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime );
```

Liste 11.2 `vPortSuppressTicksAndSleep` API işlevi prototipi (prototype)

11.3.3 eTaskConfirmSleepModeStatus İşlevi

`eTaskConfirmSleepModeStatus` API'si, uykuya (sleep) devam etmenin (proceed) uygun (ok) olup olmadığını ve süresiz olarak (indefinitely) uyumanın (sleep) uygun olup olmadığını belirlemek (determine) için uyku modu (sleep mode) durumunu döndürür (returns). Bu işlevsellik (functionality) yalnızca (only) `configUSE_TICKLESS_IDLE` 1 olarak ayarlandığında kullanılabilir (available).

```
eSleepModeStatus eTaskConfirmSleepModeStatus( void );
```

Liste 11.3 `eTaskConfirmSleepModeStatus` API işlevi prototipi

Eğer `eTaskConfirmSleepModeStatus()` işlevi `portSUPPRESS_TICKS_AND_SLEEP()` içinden çağrıldığında `eNoTasksWaitingTimeout` döndürürse, mikrodenetleyici derin uyku durumunda süresiz olarak kalabilir.

`eTaskConfirmSleepModeStatus()` yalnızca aşağıdaki (following) koşullar (conditions) doğru (true) olduğunda `eNoTasksWaitingTimeout` döndürür (return):

- Yazılım zamanlayıcıları (Software timers) kullanılmamaktadır, bu nedenle zamanlayıcının (scheduler) gelecekte herhangi bir zamanda (at any time in the future) bir zamanlayıcı geri çağırma (timer callback) işlevini yürütmesi beklenmez (not due).
- Tüm uygulama (application) görevleri ya Askıya Alınmış (Suspended) durumdadır ya da `portMAX_DELAY` zaman aşımı değeriyle (timeout value) Engellenmiş (Blocked) durumundadır, bu nedenle çizgeleyicinin (scheduler) gelecekte sabit (fixed) bir zamanda bir görevi (task) Engellenmiş durumundan çıkarması beklenmez.

Yarış koşullarını (race conditions) önlemek (a`void`) için FreeRTOS zamanlayıcısı (scheduler) `portSUPPRESS_TICKS_AND_SLEEP()` çağrılmadan önce askıya alınır (suspended) ve `portSUPPRESS_TICKS_AND_SLEEP()` tamamlandığında devam ettirilir (resumed). Bu, mikrodenetleyicinin düşük güç durumundan çıkması (exiting) ile `portSUPPRESS_TICKS_AND_SLEEP()` işlevinin yürütülmesini tamamlaması (completing) arasında uygulama (application) görevlerinin yürütülmemesini (cannot execute) sağlar (ensures). Ayrıca (Further), uyku moduna geçmenin (proceed into) uygun olduğundan emin olmak için `portSUPPRESS_TICKS_AND_SLEEP()` işlevinin zamanlayıcının (timer) durdurulması ile uyku moduna girilmesi (entered) arasında küçük (small) bir kritik bölüm (critical section) oluşturması gerekir (necessary). `eTaskConfirmSleepModeStatus()` bu kritik bölümden çağrılmalıdır.

Buna ek olarak (In addition), FreeRTOS kullanıcılara (users) `FreeRTOSConfig.h` dosyasında tanımlanan diğer iki arayüz (interface) işlevi sunar. Bu makrolar, uygulama yazarının (application writer) MCU düşük güç durumuna (low power state) yerleştirilmeden (placed) önce ve sonra sırasıyla (respectively) ek (additional) adımlar (steps) eklemesine (add) olanak tanır.

11.3.4 configPRE_SLEEP_PROCESSING yapılandırması (configuration)

```
configPRE_SLEEP_PROCESSING( xExpectedIdleTime )
```

Liste 11.4 `configPRE_SLEEP_PROCESSING` makrosu için prototip

Kullanıcı MCU'yu (MCU) düşük güç (low-power) moduna girmeden önce (before), diğer çevresel saatleri (peripheral clocks) kapatmak (turning off), sistem frekansını (frequency) azaltmak (reducing) gibi sistem güç tüketimini (power consumption) azaltacak sistem parametrelerini (parameters) yapılandırmak (configure) için `configPRE_SLEEP_PROCESSING()` çağrılmalıdır (must be called).

11.3.5 configPOST_SLEEP_PROCESSING yapılandırması

```
configPOST_SLEEP_PROCESSING( xExpectedIdleTime )
```

Liste 11.5 `configPOST_SLEEP_PROCESSING` makrosu için prototip

Düşük güç modundan (low-power mode) çıktıktan sonra (After exiting), kullanıcı (user) sistemin ana frekansını (main frequency) ve çevresel işlevlerini (peripheral functions) geri yüklemek (restore) için `configPOST_SLEEP_PROCESSING()` işlevini (function) çağırmalıdır (should call).

11.4 portSUPPRESS_TICKS_AND_SLEEP() Makrosunu Uygulama (Implementing)

Eğer kullanımda olan (in use) FreeRTOS bağlantı noktası (port) varsayılan (default) bir `portSUPPRESS_TICKS_AND_SLEEP()` uygulaması (implementation) sağlamıyorsa (does not provide), uygulama yazarı (application writer) `FreeRTOSConfig.h` içinde `portSUPPRESS_TICKS_AND_SLEEP()` tanımlayarak (defining) kendi uygulamasını (implementation) sağlayabilir (provide). Eğer kullanımda olan FreeRTOS bağlantı noktası (port) varsayılan bir `portSUPPRESS_TICKS_AND_SLEEP()` uygulaması sağlıyorsa (does provide), uygulama yazarı `FreeRTOSConfig.h` içinde `portSUPPRESS_TICKS_AND_SLEEP()` tanımlayarak (defining) varsayılan (default) uygulamayı (implementation) geçersiz kılabilir (override).

Aşağıdaki kaynak kodu (source code), bir uygulama yazarı (application writer) tarafından `portSUPPRESS_TICKS_AND_SLEEP()` işlevinin nasıl (how) uygulanabileceğine (might be implemented) dair bir örnektir (example). Örnek temeldir (basic) ve çekirdek (kernel) tarafından korunan (maintained) zaman (time) ile takvim (calendar) zamanı arasında bazı kaymalara (slippage) neden (introduce) olacaktır. Örnekte (example) gösterilen (shown) işlev çağrılarından (function calls), yalnızca `vTaskStepTick()` ve `eTaskConfirmSleepModeStatus()` FreeRTOS API'sinin bir parçasıdır (part of). Diğer işlevler (functions), kullanımda olan (in use) donanımda (hardware) mevcut (available) olan saatlere (clocks) ve güç tasarrufu modlarına (power saving modes) özgüdür (specific to) ve bu nedenle (as such) uygulama yazarı tarafından sağlanmalıdır (must be provided).

```
/* Önce portSUPPRESS_TICKS_AND_SLEEP() makrosunu tanımlayın. Parametre

   (parameter), çekirdeğin bir sonraki (next) yürütülmesi (execute) gereken

   zamana kadar tick (kene) cinsinden süredir (time). */

#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )



/* portSUPPRESS_TICKS_AND_SLEEP() tarafından çağrılan (called)

   işlevi (function) tanımlayın. */

void vApplicationSleep( TickType_t xExpectedIdleTime )

{

    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

    eSleepModeStatus eSleepStatus;



    /* Geçerli (current) zamanı, mikrodenetleyici düşük bir güç durumundayken
(state)

       çalışır durumda (operational) kalacak (remain) bir zaman

       kaynağından (time source) okuyun. */

    ulLowPowerTimeBeforeSleep = ulGetExternalTime();
```

```
    /* Tick kesmesi (interrupt) oluşturan (generating) zamanlayıcıyı (timer)
durdurun. */

    prvStopTickInterruptTimer();



    /* MCU'yu uyku modundan (sleep mode) çıkaran (bringing out) kesmeleri
(interrupts)

       etkilemeyecek (will not effect) bir kritik bölüme (critical section) girin.
*/

    disable_interrupts();



    /* Uyku (sleep) moduna girmenin hala uygun (ok) olduğundan emin olun. */

    eSleepStatus = eTaskConfirmSleepModeStatus();



    if( eSleepStatus == eAbortSleep )

    {

        /* Bu makro yürütüldüğünden (executed) beri bir görev Engellenmiş
durumundan

           (Blocked state) çıkarılmıştır (moved out) veya bir bağlam anahtarı

           (context switch) beklemede (pending) tutulmaktadır. Bir uyku

           durumuna (sleep state) girmeyin (Do not enter). Tick'i yeniden

           başlatın (Restart) ve kritik bölümden (critical section) çıkın. */

        prvStartTickInterruptTimer();

        enable_interrupts();

    }

    else

    {

        if( eSleepStatus == eNoTasksWaitingTimeout )

        {

            /* Mikrodenetleyiciyi (microcontroller) gelecekte (future) sabit
(fixed) bir zamanda

               düşük güç (low power) durumundan çıkarmak (bring out)
```

```
               için bir kesme (interrupt) yapılandırmak (configure) gerekli
değildir (not necessary). */

            prvSleep();

        }

        else

        {

            /* Çekirdeğin bir sonraki yürütülmesi (execute) gereken zamanda (time)

               mikrodenetleyiciyi düşük güç (low power) durumundan

               çıkarmak için bir kesme (interrupt) yapılandırın. Kesme,

               mikrodenetleyici düşük bir güç durumundayken çalışır

               durumda (operational) kalan (remains) bir kaynaktan (source)
oluşturulmalıdır (generated). */

            vSetWakeTimeInterrupt( xExpectedIdleTime );


            /* Düşük güç (low power) durumuna (state) girin. */

            prvSleep();


            /* Mikrodenetleyici, vSetWakeTimeInterrupt() çağrısı tarafından

               yapılandırılandan (configured) başka (other than) bir kesme

               (interrupt) ile düşük güç modundan çıkarıldıysa (brought out),

               xExpectedIdleTime'dan daha az olacak olan

               mikrodenetleyicinin gerçekte (actually) ne kadar süre (how long)

               düşük güç durumunda kaldığını (was in) belirleyin (Determine).

               portSUPPRESS_TICKS_AND_SLEEP() çağrılmadan önce çizgeleyicinin

               (scheduler) askıya alındığına (suspended) ve

               portSUPPRESS_TICKS_AND_SLEEP() döndüğünde (returns) devam
ettirildiğine (resumed)

               dikkat edin (Note). Bu nedenle bu işlev tamamlanana kadar (until)
başka

               (no other) hiçbir görev yürütülmeyecektir (will execute). */

            ulLowPowerTimeAfterSleep = ulGetExternalTime();
```

```
            /* Çekirdeklerin tick sayısını (tick count), mikrodenetleyicinin

               düşük güç durumunda geçirdiği zamanı (time spent)

               hesaba katacak (account for) şekilde düzeltin (Correct). */

            vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );

        }



        /* Kritik bölümden (critical section) çıkın (Exit) - bunu (this)
prvSleep()

           çağrılarından (calls) hemen sonra (immediately after) yapmak mümkün
(possible) olabilir (might be). */

        enable_interrupts();



        /* Tick kesmesi (interrupt) üreten (generating) zamanlayıcıyı (timer)
yeniden başlatın (Restart). */

        prvStartTickInterruptTimer();

    }

}
```

Liste 11.6 Kullanıcı (user) tanımlı bir `portSUPPRESS_TICKS_AND_SLEEP()` uygulaması (implementation) örneği

11.5 Boşta (Idle) Görev Kancası (Hook) İşlevi

Boşta (Idle) görevi (task), isteğe bağlı (optionally) olarak uygulama (application) tanımlı bir kanca (hook) (veya geri çağırma - callback) işlevini - boşta kancasını (idle hook) çağırabilir (can call). Boşta görevi en düşük öncelikte (lowest priority) çalışır (runs), bu nedenle böyle bir boşta kanca işlevi (idle hook function) yalnızca (only) çalışabilen (able to run) daha yüksek (higher) öncelikli hiçbir görev olmadığında yürütülür (get executed). Bu, Boşta kancası (Idle hook) işlevini işlemciyi düşük güç durumuna (low power state) getirmek (put) için ideal bir yer (ideal place) haline getirir (makes) - gerçekleştirilecek (performed) hiçbir işlem (processing) olmadığında (whenever) otomatik (automatic) bir güç tasarrufu (power saving) sağlar (providing). Boşta (Idle) kancası (hook), yalnızca `FreeRTOSConfig.h` içinde `configUSE_IDLE_HOOK` 1 olarak ayarlanmışsa (set to) çağrılır (called).

```
void vApplicationIdleHook( void );
```

Liste 11.7 `vApplicationIdleHook` API işlevi prototipi (prototype)

Boşta (idle) görevi (task) çalıştığı (running) sürece (as long as) boşta kancası (idle hook) tekrar tekrar (repeatedly) çağrılır. Boşta kanca işlevinin engellenmesına (block) neden (cause) olabilecek hiçbir API işlevini çağırmaması son derece önemlidir (paramount). Ayrıca (Also), eğer uygulama (application) `vTaskDelete()` API işlevinden yararlanıyorsa (makes use of), boşta görev kancasının periyodik olarak dönmesine (periodically return) izin verilmelidir, çünkü (since) boşta görevi (idle task), silinmiş olan (deleted) göreve RTOS çekirdeği (kernel) tarafından tahsis edilen (allocated) kaynakları (resources) temizlemekten (cleaning up) sorumludur (responsible for).

Bölüm 12

Geliştirici Desteği (Developer Support)

12.1 Giriş

Bu bölüm, aşağıdakileri yaparak üretkenliği (productivity) en üst düzeye çıkarmak (maximize) için dahil edilen (included) bir dizi (set of) özelliği (features) vurgulamaktadır (highlights):

- Bir uygulamanın nasıl davrandığına (behaving) dair içgörü (insight) sağlamak (Providing).
- Optimizasyon için fırsatları (opportunities) vurgulamak (Highlighting).
- Hataları meydana geldikleri (occur) noktada yakalamak (Trapping).

12.2 configASSERT()

C dilinde `assert()` makrosu, program tarafından yapılan bir iddiayı (assertion - bir varsayımı - assumption) doğrulamak (verify) için kullanılır. İddia (assertion) bir C ifadesi (expression) olarak yazılır ve ifadenin sonucu yanlış (false - 0) olarak değerlendirilirse (evaluates), iddianın başarısız (failed) olduğu kabul edilir (deemed). Örneğin, Liste 12.1 `pxMyPointer` işaretçisinin (pointer) `NULL` olmadığı iddiasını test eder.

```
/* pxMyPointer'ın NULL olmadığı iddiasını (assertion) test edin */

assert( pxMyPointer != NULL );
```

Liste 12.1 `pxMyPointer`'ın `NULL` olmadığını kontrol etmek (check) için standart C `assert()` makrosunun kullanılması

Uygulama yazarı (application writer), `assert()` makrosunun bir uygulamasını (implementation) sağlayarak bir iddianın (assertion) başarısız olması durumunda (fails) gerçekleştirilecek (to take) eylemi (action) belirtir.

FreeRTOS kaynak kodu `assert()` öğesini çağırmaz (does not call), çünkü `assert()`, FreeRTOS'un derlendiği (compiled) tüm derleyicilerde (compilers) mevcut (available) değildir. Bunun yerine (Instead), FreeRTOS kaynak kodu `configASSERT()` adı verilen (called) bir makroya çok sayıda (lots of) çağrı içerir; bu makro uygulama yazarı tarafından `FreeRTOSConfig.h` içinde tanımlanabilir (defined) ve tam olarak (exactly) standart C `assert()` gibi davranır (behaves).

Başarısız (failed) bir iddia (assertion) ölümcül bir hata (fatal error) olarak ele alınmalıdır (treated as). Bir iddiada başarısız olmuş bir satırı geçerek yürütmeye (execute past a line) çalışmayın (Do not attempt).

`configASSERT()` kullanmak, en yaygın hata kaynaklarının çoğunu anında yakalayarak (trapping) ve tanımlayarak (identifying) üretkenliği (productivity) artırır. Bir FreeRTOS uygulaması geliştirirken (developing) veya hata ayıklarken (debugging) `configASSERT()` tanımlanmış olması şiddetle tavsiye edilir (strongly advised).

`configASSERT()` işlevinin tanımlanması (Defining), çalışma zamanı (run-time) hata ayıklamasında (debugging) büyük ölçüde yardımcı (assist) olacaktır, ancak aynı zamanda (also) uygulama kod boyutunu (size) artıracak ve bu nedenle yürütülmesini yavaşlatacaktır (slow down). `configASSERT()` tanımı sağlanmazsa, varsayılan (default) boş tanım (empty definition) kullanılacak ve `configASSERT()` çağrılarının tamamı C önişlemcisi (preprocessor) tarafından tamamen kaldırılacaktır (completely removed).

12.2.1 Örnek configASSERT() tanımları (definitions)

Liste 12.2'de gösterilen `configASSERT()` tanımı, bir uygulama bir hata ayıklayıcının (debugger) kontrolü altında yürütülürken (executed) yararlıdır (useful). Bir iddiada başarısız olan herhangi bir satırda (any line) yürütmeyi durduracaktır (halt), böylece (so) iddiada başarısız olan satır, hata ayıklama oturumu (debug session) duraklatıldığında (paused) hata ayıklayıcı (debugger) tarafından görüntülenen satır olacaktır.

```
/* Tick kesmesinin (interrupt) yürütülmesini durdurması (stops executing) için
kesmeleri (interrupts)

   devre dışı bırakın (Disable), ardından yürütmenin (execution) iddiada başarısız
olan (failed) satırı

   geçmemesi için bir döngüye oturtun (sit in a loop). Eğer donanım

   bir hata ayıklama kesme (debug break) komutunu (instruction) destekliyorsa
(supports), for() döngüsü

   yerine (in place of) hata ayıklama kesme komutu kullanılabilir. */

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for(;;); }
```

Liste 12.2 Bir hata ayıklayıcının kontrolü altında yürütülürken (executing under the control of a debugger) faydalı olan basit (simple) bir `configASSERT()` tanımı

Liste 12.3'te gösterilen `configASSERT()` tanımı, bir uygulama bir hata ayıklayıcının kontrolü altında yürütülmediğinde kullanışlıdır (useful). Başarısız bir iddiayı (assertion) içeren kaynak kodu (source code) satırını yazdırır (prints out) veya bir şekilde kaydeder (records). İddianın başarısız olduğu satır, kaynak dosyasının adını elde etmek (obtain) için standart C `__FILE__` makrosu ve kaynak dosyasındaki satır numarasını elde etmek için standart C `__LINE__` makrosu kullanılarak tanımlanır (identified).

```
/* Bu işlev FreeRTOSConfig.h başlık (header) dosyasında değil,

   bir C kaynak dosyasında (source file) tanımlanmalıdır. */

void vAssertCalled( const char *pcFile, uint32_t ulLine )

{

    /* Bu işlevin içinde (Inside), pcFile hatayı algılayan (detected) satırı
içeren

       kaynak dosyasının (source file) adını tutar ve ulLine kaynak dosyasındaki

       satır numarasını tutar. pcFile ve ulLine değerleri yazdırılabilir

       (printed out) veya aşağıdaki (following) sonsuz döngüye (infinite loop)

       girilmeden (entered) önce kaydedilebilir. */

    RecordErrorInformationHere( pcFile, ulLine );



    /* Tick kesmesinin yürütülmesini durdurması için kesmeleri devre dışı bırakın,
ardından

       yürütmenin (execution) iddiada başarısız olan satırı geçmemesi

       için bir döngüde bekletin (sit in a loop). */

    taskDISABLE_INTERRUPTS();

    for( ;; );

}

/*-----------------------------------------------------------*/



/* Aşağıdaki iki satır FreeRTOSConfig.h dosyasına yerleştirilmelidir (placed). */

extern void vAssertCalled( const char *pcFile, unsigned long ulLine );

#define configASSERT( x ) if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )
```

Liste 12.3 Başarısız bir iddiayı içeren kaynak kodu satırını kaydeden (records) bir `configASSERT()` tanımı

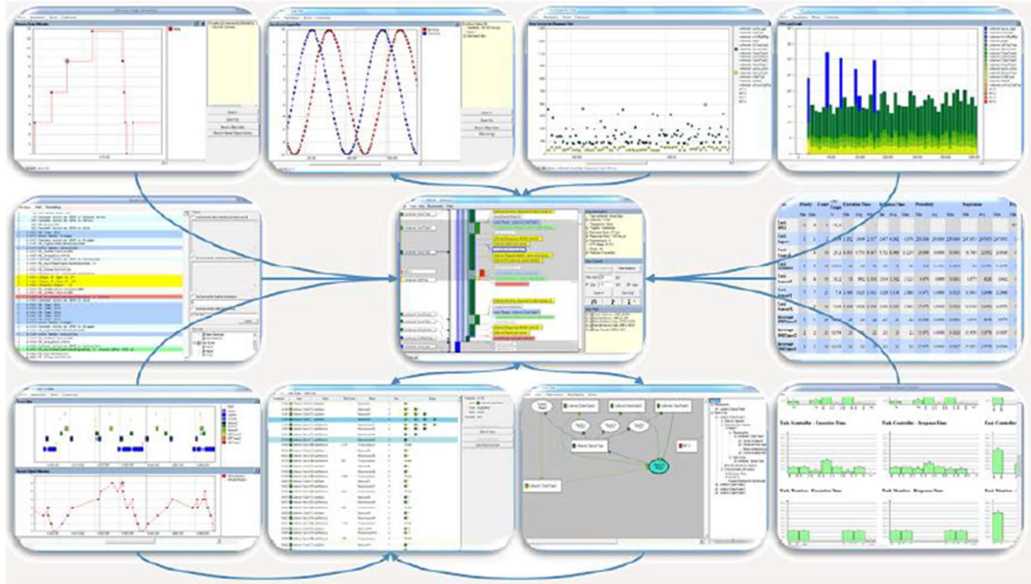
12.3 FreeRTOS için Tracealyzer

FreeRTOS için Tracealyzer, ortak şirketimiz (partner company) Percepio tarafından sağlanan çalışma zamanlı (run-time) bir tanılama (diagnostic) ve optimizasyon aracıdır (tool).

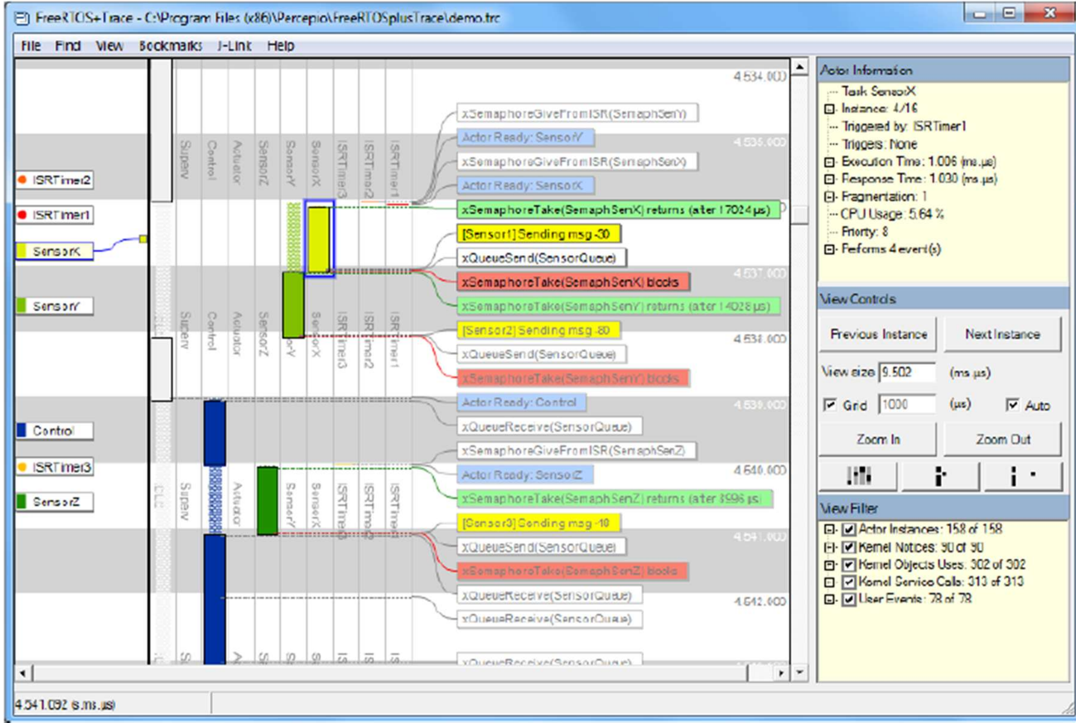
FreeRTOS için Tracealyzer değerli (valuable) dinamik (dynamic) davranış (behavior) bilgilerini yakalar (captures), ardından yakalanan bilgileri birbirine bağlı (interconnected) grafiksel görünümlerde (graphical views) sunar (presents). Araç ayrıca birden fazla (multiple) senkronize görünümü görüntüleme yeteneğine de sahiptir (capable of).

Yakalanan (captured) bilgiler, bir FreeRTOS uygulamasını analiz ederken (analyzing), sorun giderirken (troubleshooting) veya basitçe optimize ederken paha biçilmezdir (invaluable).

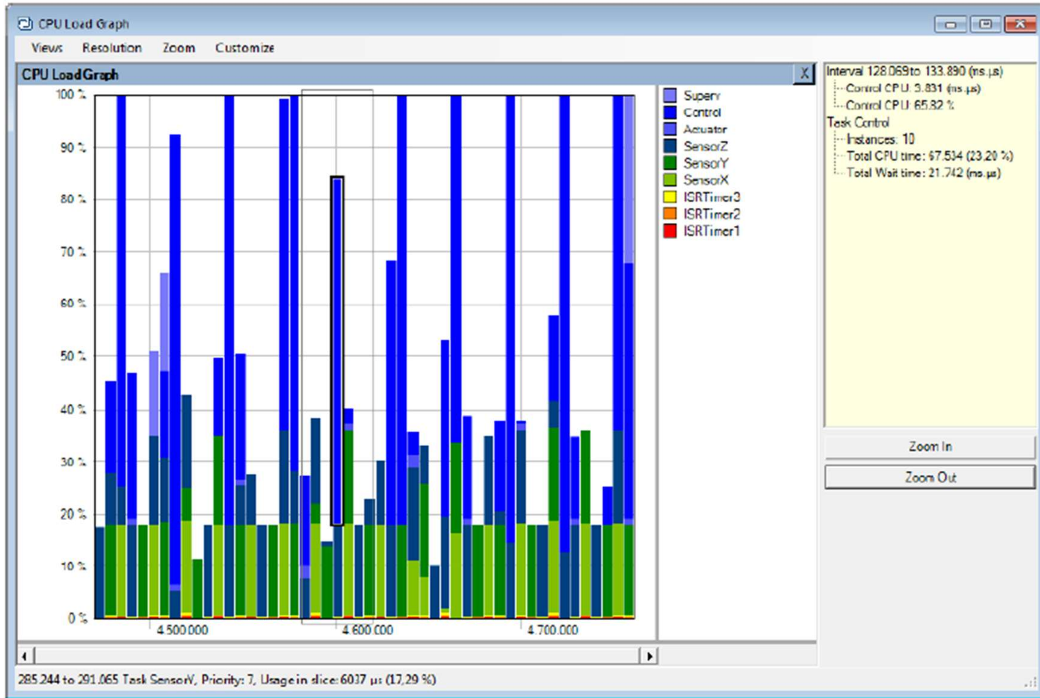
FreeRTOS için Tracealyzer, geleneksel (traditional) bir hata ayıklayıcı (debugger) ile yan yana (side-by-side) kullanılabilir ve hata ayıklayıcının görünümünü (view) daha yüksek seviyeli (higher level), zamana dayalı (time-based) bir perspektifle (perspective) tamamlar (complements).



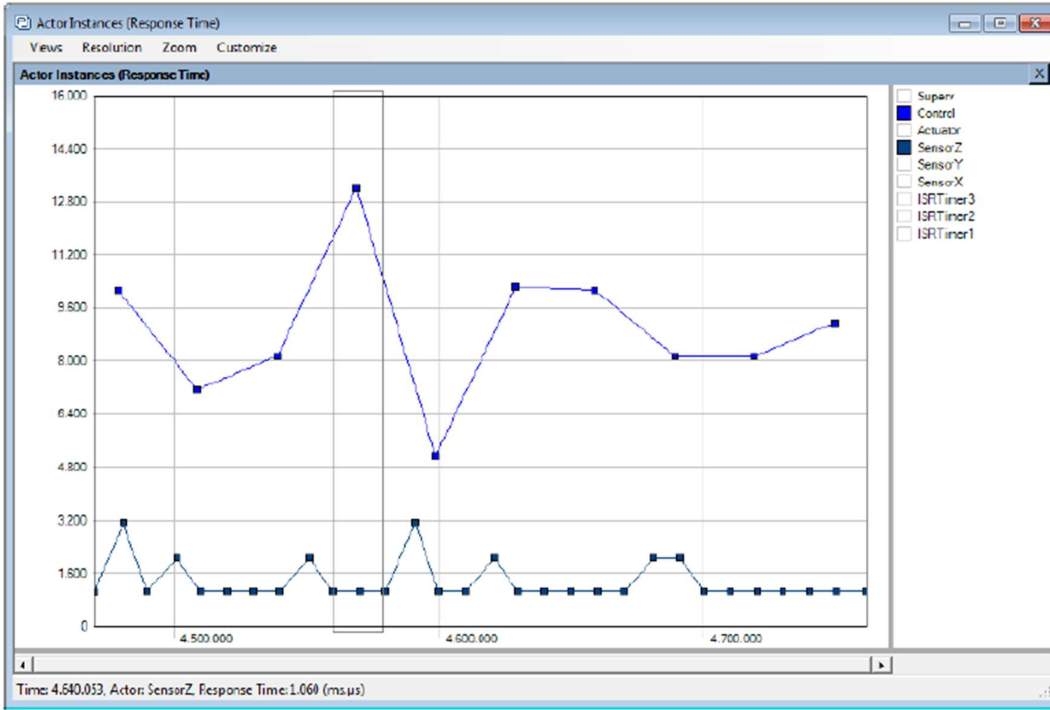
Şekil 12.1 FreeRTOS+Trace 20'den fazla birbirine bağlı (interconnected) görünüm (views) içerir



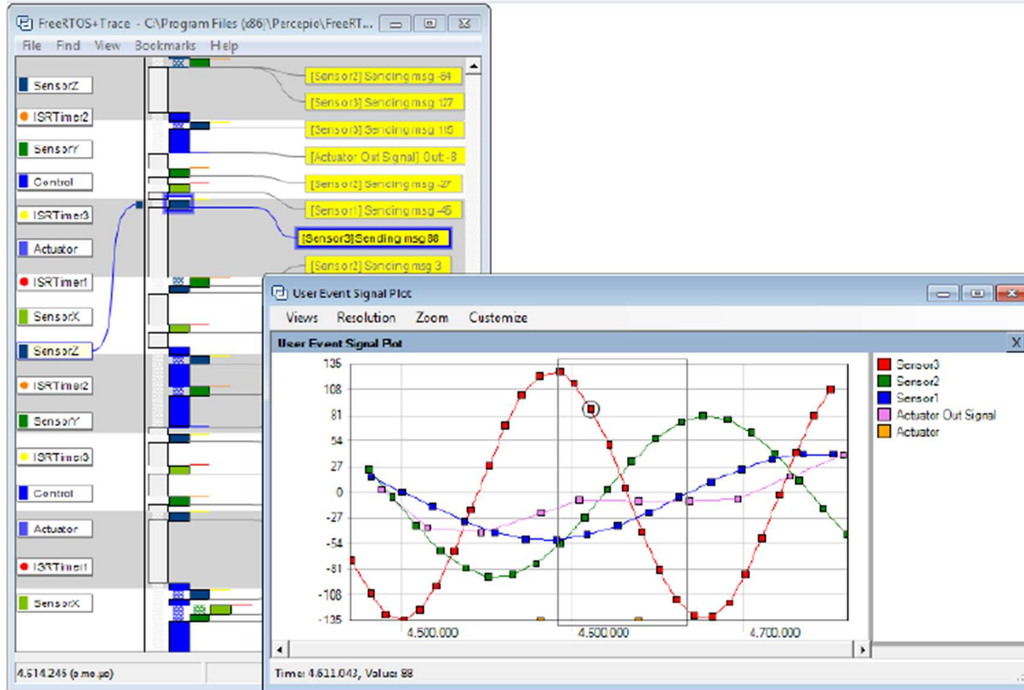
Şekil 12.2 FreeRTOS+Trace ana iz (trace) görünümü - 20'den fazla birbirine bağlı iz (trace) görünümünden biri



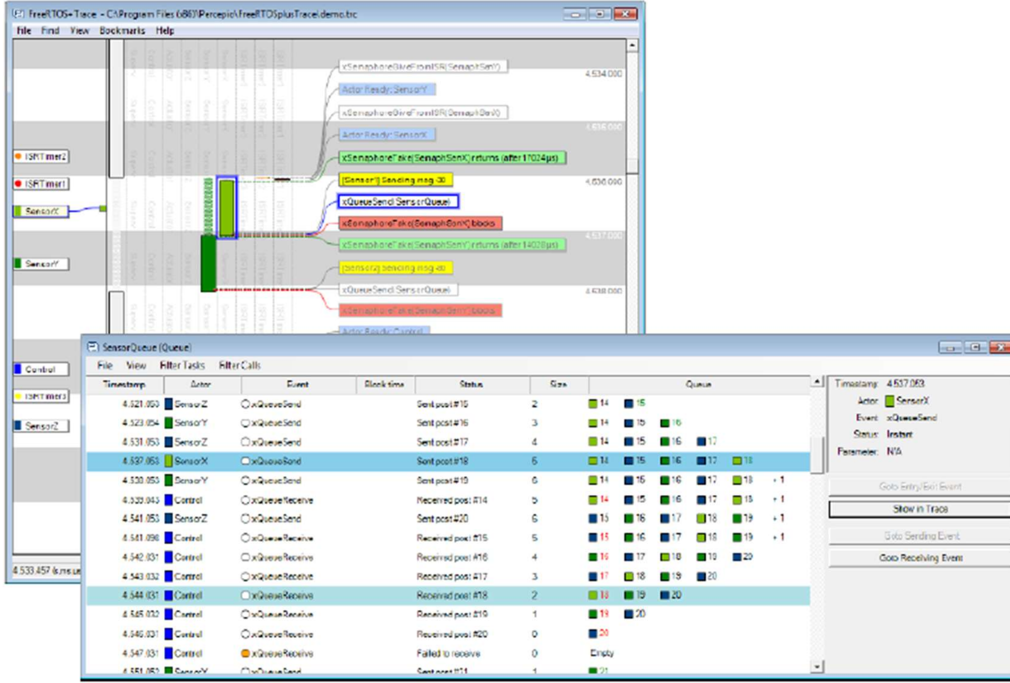
Şekil 12.3 FreeRTOS+Trace CPU yük (load) görünümü - 20'den fazla birbirine bağlı iz görünümünden biri



Şekil 12.4 FreeRTOS+Trace tepki süresi (response time) görünümü - 20'den fazla birbirine bağlı iz görünümünden biri



Şekil 12.5 FreeRTOS+Trace kullanıcı (user) olay grafiği (event plot) görünümü - 20'den fazla birbirine bağlı iz görünümünden biri



Şekil 12.6 FreeRTOS+Trace çekirdek nesne geçmişi (kernel object history) görünümü - 20'den fazla birbirine bağlı iz görünümünden biri

12.4 Hata Ayıklama (Debug) İle İlgili Kanca (Geri Çağırma - Callback) İşlevleri

12.4.1 Malloc başarısız (failed) kancası (hook)

Malloc başarısız (failed) kancası (veya geri çağırması - callback) Bölüm 3, Yığın Bellek Yönetimi'nde (Heap Memory Management) açıklanmıştı (described).

Bir malloc başarısız kancasının tanımlanması, bir görev (task), kuyruk (queue), semafor (semaphore) veya olay grubu (event group) oluşturma girişiminin (attempt) başarısız olması durumunda (fails) uygulama geliştiricisinin (developer) derhal (immediately) bilgilendirilmesini (notified) sağlar (ensures).

12.4.2 Yığın taşması (Stack overflow) kancası

Yığın taşması (Stack overflow) kancasının ayrıntıları (details) Bölüm 13.3, Yığın Taşması'nda (Stack Overflow) verilmiştir (provided).

Bir yığın taşması kancası tanımlamak (Defining), bir görev tarafından kullanılan yığın (stack) miktarının (amount) göreve tahsis edilen (allocated) yığın alanını (stack space) aşması (exceeds) durumunda uygulama geliştiricisinin bilgilendirilmesini sağlar.

12.5 Çalışma Zamanı ve Görev Durum Bilgilerini Görüntüleme

12.5.1 Görev Çalışma Zamanı İstatistikleri

Görev çalışma zamanı istatistikleri, her bir görevin aldığı işlemci süresinin miktarı hakkında bilgi sağlar. Bir görevin çalışma zamanı, uygulama başlatıldığından beri görevin Çalışıyor (Running) durumunda geçirdiği toplam süredir.

Çalışma zamanı istatistikleri, bir projenin geliştirme aşamasında profil çıkarma ve hata ayıklama yardımcısı olarak kullanılmak üzere tasarlanmıştır. Sağladıkları bilgiler, yalnızca çalışma zamanı istatistik saati olarak kullanılan sayaç taşana kadar geçerlidir. Çalışma zamanı istatistiklerinin toplanması, görev bağlam anahtarı süresini artıracaktır.

İkili (binary) çalışma zamanı istatistik bilgilerini elde etmek için `uxTaskGetSystemState()` API işlevini çağırın. Çalışma zamanı istatistik bilgilerini insan tarafından okunabilir bir ASCII tablosu olarak elde etmek için `vTaskGetRunTimeStatistics()` yardımcı işlevini çağırın.

12.5.2 Çalışma Zamanı İstatistik Saati

Çalışma zamanı istatistiklerinin, bir tick periyodunun kesirlerini ölçmesi gerekir. Bu nedenle RTOS tick sayacı çalışma zamanı istatistik saati olarak kullanılmaz ve bunun yerine saat uygulama kodu tarafından sağlanır. Çalışma zamanı istatistik saatinin frekansının, tick kesmesi frekansından 10 ila 100 kat daha hızlı olması önerilir. Çalışma zamanı istatistik saati ne kadar hızlı olursa, istatistikler o kadar doğru olur, ancak zaman değeri de o kadar çabuk taşar.

İdeal olarak, zaman değeri başka bir işleme yükü olmadan okunabilen serbest çalışan 32 bitlik bir çevresel zamanlayıcı/sayaç tarafından üretilecektir. Mevcut çevresel birimler ve saat hızları bu tekniği mümkün kılmıyorsa, alternatif ancak daha az verimli teknikler şunlardır:

• İstenen çalışma zamanı istatistik saati frekansında periyodik kesme üretmek için bir çevresel birim yapılandırın ve ardından çalışma zamanı istatistik saati olarak üretilen kesme sayısını kullanın. Bu yöntem, periyodik kesme yalnızca çalışma zamanı istatistik saati sağlama amacıyla kullanılıyorsa çok verimsizdir. Ancak, uygulama zaten uygun frekansta bir periyodik kesme kullanıyorsa, mevcut kesme hizmet rutinine üretilen kesme sayısının eklenmesi basit ve verimlidir.

• Serbest çalışan 16 bitlik çevresel zamanlayıcının mevcut değerini 32 bitlik değerin en az anlamlı 16 biti olarak ve zamanlayıcının taşma sayısını 32 bitlik değerin en anlamlı 16 biti olarak kullanarak bir 32 bitlik değer üretin.

ARM Cortex-M SysTick zamanlayıcısının mevcut değerini RTOS tick sayısıyla birleştirerek, uygun ve biraz karmaşık bir manipülasyonla çalışma zamanı istatistik saati üretmek mümkündür. FreeRTOS indirmesindeki demo projelerinden bazıları bunun nasıl başarıldığını göstermektedir.

12.5.3 Çalışma Zamanı İstatistiklerini Toplamak İçin Uygulamayı Yapılandırma

Aşağıda görev çalışma zamanı istatistiklerini toplamak için gerekli makrolar hakkında ayrıntılar verilmektedir. Başlangıçta makroların RTOS bağlantı noktası katmanına dahil edilmesi amaçlanmıştı — bu nedenle makrolar 'port' ön ekine

sahiptir — ancak bunları `FreeRTOSConfig.h` içinde tanımlamanın daha pratik olduğu kanıtlanmıştır.

Çalışma zamanı istatistiklerinin toplanmasında kullanılan makrolar:

`configGENERATE_RUN_TIME_STATS` — Bu makro `FreeRTOSConfig.h` içinde 1 olarak ayarlanmalıdır. Bu makro 1 olarak ayarlandığında, çizgeleyici bu bölümde ayrıntılı olarak açıklanan diğer makroları uygun zamanlarda çağıracaktır.

`portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` — Bu makro, çalışma zamanı istatistik saatini sağlamak için kullanılan çevresel birimi başlatmak üzere sağlanmalıdır.

`portGET_RUN_TIME_COUNTER_VALUE()` veya `portALT_GET_RUN_TIME_COUNTER_VALUE(Time)` — Bu iki makrodan biri, mevcut çalışma zamanı istatistik saati değerini döndürmek için sağlanmalıdır. Bu, uygulamanın ilk önyüklemesinden bu yana çalışma zamanı istatistik saati birimlerinde toplam çalışma süresidir. İlk makro kullanılırsa, mevcut saat değerine göre değerlendirilecek şekilde tanımlanmalıdır. İkinci makro kullanılırsa, 'Time' parametresini mevcut saat değerine ayarlayacak şekilde tanımlanmalıdır.

12.5.4 uxTaskGetSystemState() API İşlevi

`uxTaskGetSystemState()` FreeRTOS çizgeleyicisi kontrolündeki her görev için durum bilgisi anlık görüntüsü sağlar. Bilgi, dizideki her görev için bir dizin içeren `TaskStatus_t` yapıları dizisi olarak sunulur. `TaskStatus_t`, Liste 12.5 ve aşağıda açıklanmaktadır.

```
UBaseType_t uxTaskGetSystemState( TaskStatus_t * const pxTaskStatusArray,
const UBaseType_t uxArraySize,
configRUN_TIME_COUNTER_TYPE * const pulTotalRunTime );
```

Liste 12.4 uxTaskGetSystemState() API işlevi prototipi

Not: `configRUN_TIME_COUNTER_TYPE` geriye dönük uyumluluk için varsayılan olarak `uint32_t`'dir, ancak `uint32_t` çok kısıtlayıcıysa `FreeRTOSConfig.h` içinde geçersiz kılınabilir.

uxTaskGetSystemState() parametreleri ve dönüş değeri:

`pxTaskStatusArray` — `TaskStatus_t` yapıları dizisine işaretçi.

Dizi, her görev için en az bir `TaskStatus_t` yapısı içermelidir. Görev sayısı `uxTaskGetNumberOfTasks()` API işlevi kullanılarak belirlenebilir. `TaskStatus_t` yapısı Liste 12.5'te gösterilmiştir ve `TaskStatus_t` yapı üyeleri sonraki listede açıklanmaktadır.

`uxArraySize` — `pxTaskStatusArray` parametresi tarafından işaret edilen dizinin boyutu. Boyut, dizideki dizin sayısı (dizide bulunan `TaskStatus_t` yapılarının sayısı) olarak belirtilir, dizideki bayt sayısı olarak değil.

`pulTotalRunTime` — `configGENERATE_RUN_TIME_STATS` `FreeRTOSConfig.h` içinde 1 olarak ayarlanmışsa, *`pulTotalRunTime` `uxTaskGetSystemState()` tarafından hedef önyükleme yaptığından bu yana toplam çalışma süresine (uygulama tarafından sağlanan çalışma zamanı istatistik saati tarafından tanımlandığı şekilde) ayarlanır. `pulTotalRunTime` isteğe bağlıdır ve toplam çalışma süresi gerekmiyorsa `NULL` olarak ayarlanabilir.

Dönüş değeri — `uxTaskGetSystemState()` tarafından doldurulan `TaskStatus_t` yapılarının sayısı döndürülür. Döndürülen değer, `uxTaskGetNumberOfTasks()` API işlevi tarafından döndürülen sayıya eşit olmalıdır, ancak `uxArraySize` parametresinde geçirilen değer çok küçükse sıfır olacaktır.

```
typedef struct xTASK_STATUS
{
    TaskHandle_t xHandle;
    const char *pcTaskName;
    UBaseType_t xTaskNumber;
    eTaskState eCurrentState;
    UBaseType_t uxCurrentPriority;
    UBaseType_t uxBasePriority;
    configRUN_TIME_COUNTER_TYPE ulRunTimeCounter;
    StackType_t * pxStackBase;
    #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS == 1 )
)
        StackType_t * pxTopOfStack;
        StackType_t * pxEndOfStack;
    #endif
    uint16_t usStackHighWaterMark;
    #if ( ( configUSE_CORE_AFFINITY == 1 ) && ( configNUMBER_OF_CORES > 1 ) )
        UBaseType_t uxCoreAffinityMask;
    #endif
} TaskStatus_t;
```

Liste 12.5 TaskStatus_t yapısı

`TaskStatus_t` yapı üyeleri:

`xHandle` — Yapıdaki bilgilerin ilişkili olduğu görevin tanıtıcısıdır.

`pcTaskName` — Görevin insan tarafından okunabilir metin adıdır.

`xTaskNumber` — Her görevin benzersiz bir `xTaskNumber` değeri vardır. Bir uygulama çalışma zamanında görevler oluşturup silerse, bir görevin daha önce silinen bir görevle aynı tanıtıcıya sahip olması mümkündür. `xTaskNumber` uygulama kodunun ve çekirdek farkında hata ayıklayıcılarının hâlâ geçerli olan bir görevi, geçerli görevle aynı tanıtıcıya sahip silinmiş bir görevden ayırt etmesine olanak tanır.

`eCurrentState` — Görevin durumunu tutan numaralandırılmış tür. `eCurrentState` şu değerlerden biri olabilir: `eRunning`, `eReady`, `eBlocked`, `eSuspended`, `eDeleted`. Bir görev, yalnızca `vTaskDelete()` çağrısıyla silindiği zaman ile Boşta görevi silinen görevin iç veri yapıları ve yığını için ayrılan belleği serbest bıraktığı zaman arasındaki

kısa süre boyunca `eDeleted` durumunda raporlanacaktır. Bu süreden sonra görev artık hiçbir şekilde var olmayacak ve tanıtıcısını kullanmaya çalışmak geçersizdir.

`uxCurrentPriority` — `uxTaskGetSystemState()` çağrıldığı andaki görevin önceliğidir. `uxCurrentPriority` yalnızca, Bölüm 8.3 Mutex'ler (ve İkili Semaforlar) bölümünde açıklanan öncelik miras mekanizmasına uygun olarak göreve geçici olarak daha yüksek bir öncelik atanmışsa uygulama geliştiricisi tarafından atanan öncelikten daha yüksek olacaktır.

`uxBasePriority` — Uygulama geliştiricisi tarafından göreve atanan öncelik. `uxBasePriority` yalnızca `configUSE_MUTEXES` `FreeRTOSConfig.h` içinde 1 olarak ayarlandığında geçerlidir.

`ulRunTimeCounter` — Görev oluşturulduğundan bu yana görev tarafından kullanılan toplam çalışma zamanı. Toplam çalışma zamanı, çalışma zamanı istatistiklerinin toplanması için uygulama geliştiricisi tarafından sağlanan saati kullanan mutlak süre olarak sunulur. `ulRunTimeCounter` yalnızca `configGENERATE_RUN_TIME_STATS` `FreeRTOSConfig.h` içinde 1 olarak ayarlandığında geçerlidir.

`pxStackBase` — Bu göreve ayrılan yığın bölgesinin taban adresini gösterir.

`pxTopOfStack` — Bu göreve ayrılan yığın bölgesinin mevcut üst adresini gösterir. `pxTopOfStack` alanı yalnızca yığın yukarı doğru büyüyorsa (yani `portSTACK_GROWTH` sıfırdan büyükse) veya `configRECORD_STACK_HIGH_ADDRESS` `FreeRTOSConfig.h` içinde 1 olarak ayarlandığında geçerlidir.

`pxEndOfStack` — Bu göreve ayrılan yığın bölgesinin son adresini gösterir. `pxEndOfStack` alanı yalnızca yığın yukarı doğru büyüyorsa (yani `portSTACK_GROWTH` sıfırdan büyükse) veya `configRECORD_STACK_HIGH_ADDRESS` `FreeRTOSConfig.h` içinde 1 olarak ayarlandığında geçerlidir.

`usStackHighWaterMark` — Görevin yığın yüksek su işareti. Bu, görev oluşturulduğundan beri görev için kalan minimum yığın alanı miktarıdır. Görevin yığınını taşırmaya ne kadar yaklaştığının bir göstergesidir; bu değer sıfıra ne kadar yakınsa, görev yığınını taşırmaya o kadar yaklaşmıştır. `usStackHighWaterMark` bayt cinsinden belirtilir.

`uxCoreAffinityMask` — Görevin üzerinde çalışabileceği çekirdekleri gösteren bit düzeyinde bir değer. Çekirdekler 0'dan `configNUMBER_OF_CORES` - 1'e kadar numaralandırılır. Örneğin, çekirdek 0 ve çekirdek 1 üzerinde çalışabilen bir görev `uxCoreAffinityMask` değerini 0x03 olarak ayarlamış olacaktır. `uxCoreAffinityMask` alanı yalnızca `configUSE_CORE_AFFINITY` 1 olarak ve `configNUMBER_OF_CORES` 1'den büyük olarak ayarlandığında `FreeRTOSConfig.h` içinde kullanılabilir.

12.5.5 vTaskListTasks() Yardımcı İşlevi

`vTaskListTasks()` `uxTaskGetSystemState()` tarafından sağlanana benzer görev durum bilgisi sağlar, ancak bilgiyi ikili değerler dizisi yerine insan tarafından okunabilir bir ASCII tablosu olarak sunar.

`vTaskListTasks()` çok yoğun işlemci kullanan bir işlevdir ve çizgeleyiciyi uzun bir süre askıya alır. Bu nedenle işlevin yalnızca hata ayıklama amacıyla kullanılması ve üretim gerçek zamanlı sisteminde kullanılmaması önerilir.

`vTaskListTasks()` `configUSE_TRACE_FACILITY` 1 olarak ve `configUSE_STATS_FORMATTING_FUNCTIONS` `FreeRTOSConfig.h` içinde 0'dan büyük olarak ayarlandığında kullanılabilir.

```
void vTaskListTasks( char * pcWriteBuffer, size_t uxBufferLength );
```

Liste 12.6 vTaskListTasks() API işlevi prototipi

`vTaskListTasks()` parametreleri:

`pcWriteBuffer` — Formatlanmış ve insan tarafından okunabilir tablonun yazıldığı karakter arabelleğine işaretçi. Bu arabelleğin oluşturulan raporu içerecek kadar büyük olduğu varsayılır. Görev başına yaklaşık 40 bayt yeterli olmalıdır.

`uxBufferLength` — `pcWriteBuffer`'ın uzunluğu.

`vTaskListTasks()` tarafından üretilen çıktının bir örneği Şekil 12.7'de gösterilmektedir. Çıktıda: Her satır tek bir görev hakkında bilgi sağlar. İlk sütun görev adıdır. İkinci sütun görevin durumudur; burada 'X' Çalışıyor, 'R' Hazır, 'B' Engellenmiş, 'S' Askıya Alınmış ve 'D' görevin silindiği anlamına gelir. Üçüncü sütun görevin önceliğidir. Dördüncü sütun görevin yığın yüksek su işaretidir. Beşinci sütun göreve atanan benzersiz numaradır.

Not: vTaskListTasks işlevinin eski sürümü vTaskList'tir. vTaskList, `pcWriteBuffer`'ın `configSTATS_BUFFER_MAX_LENGTH` uzunluğunda olduğunu varsayar. Bu işlev yalnızca geriye dönük uyumluluk için mevcuttur. Yeni uygulamaların vTaskListTasks kullanması ve `pcWriteBuffer` uzunluğunu açıkça belirtmesi önerilir.

```
void vTaskList( signed char *pcWriteBuffer );
```

Liste 12.7 vTaskList() API işlevi prototipi

`vTaskList()` parametreleri:

`pcWriteBuffer` — Formatlanmış ve insan tarafından okunabilir tablonun yazıldığı bir karakter arabelleğine işaretçi. Arabellek, hiçbir sınır denetimi yapılmadığı için tüm tabloyu tutacak kadar büyük olmalıdır.

12.5.6 vTaskGetRunTimeStatistics() Yardımcı İşlevi

`vTaskGetRunTimeStatistics()` toplanan çalışma zamanı istatistiklerini insan tarafından okunabilir bir ASCII tablosuna biçimlendirir.

`vTaskGetRunTimeStatistics()` çok yoğun işlemci kullanan bir işlevdir ve çizgeleyiciyi uzun bir süre askıya alır. Bu nedenle işlevin yalnızca hata ayıklama amacıyla kullanılması ve üretim gerçek zamanlı sisteminde kullanılmaması önerilir.

`vTaskGetRunTimeStatistics()` `configGENERATE_RUN_TIME_STATS` 1 olarak, `configUSE_STATS_FORMATTING_FUNCTIONS` 0'dan büyük olarak ve

configUSE_TRACE_FACILITY FreeRTOSConfig.h içinde 1 olarak ayarlandığında kullanılabilir.

```
void vTaskGetRunTimeStatistics( char * pcWriteBuffer, size_t
uxBufferLength );
```

Liste 12.8 vTaskGetRunTimeStatistics() API işlevi prototipi

vTaskGetRunTimeStatistics() parametreleri:

pcWriteBuffer — Formatlanmış ve insan tarafından okunabilir tablonun yazıldığı karakter arabelleğine işaretçi. Bu arabelleğin oluşturulan raporu içerecek kadar büyük olduğu varsayılır. Görev başına yaklaşık 40 bayt yeterli olmalıdır.

uxBufferLength — pcWriteBuffer'ın uzunluğu.

vTaskGetRunTimeStatistics() tarafından üretilen çıktının bir örneği Şekil 12.8'de gösterilmektedir. Çıktıda: Her satır tek bir görev hakkında bilgi sağlar. İlk sütun görev adıdır. İkinci sütun görevin Çalışıyor durumunda geçirdiği süre miktarıdır (mutlak değer olarak). Üçüncü sütun görevin Çalışıyor durumunda geçirdiği süre miktarını hedef önyükleme yapıldığından bu yana toplam sürenin yüzdesi olarak gösterir.

```
PolSEM1     994      <1%
PolSEM2     23248    1%
GenQ        194479   16%
MuLow       3690     <1%
Rec3        229450   18%
CNT1        242720   19%
PeekL       94       <1%
CNT_INC     165      <1%
CNT2        243166   20%
SUSP_RX     243192   20%
IDLE        55       <1%
```

Figure 12.8 *Example output generated by vTaskGetRunTimeStatistics()*

Not: vTaskGetRunTimeStatistics işlevinin eski sürümü vTaskGetRunTimeStats'tır. vTaskGetRunTimeStats, pcWriteBuffer'ın configSTATS_BUFFER_MAX_LENGTH uzunluğunda olduğunu varsayar. Bu işlev yalnızca geriye dönük uyumluluk için mevcuttur. Yeni uygulamaların vTaskGetRunTimeStatistics kullanması ve pcWriteBuffer uzunluğunu açıkça belirtmesi önerilir.

```
void vTaskGetRunTimeStats( signed char *pcWriteBuffer );
```

Liste 12.9 vTaskGetRunTimeStats() API işlevi prototipi

vTaskGetRunTimeStats() parametreleri:

`pcWriteBuffer` — Formatlanmış ve insan tarafından okunabilir tablonun yazıldığı bir karakter arabelleğine işaretçi. Arabellek, hiçbir sınır denetimi yapılmadığı için tüm tabloyu tutacak kadar büyük olmalıdır.

12.5.7 Çalışma Zamanı İstatistikleri Oluşturma ve Görüntüleme, Çalışılmış Bir Örnek

Bu örnek, 32 bitlik bir çalışma zamanı istatistik saati üretmek için varsayımsal bir 16 bitlik zamanlayıcı kullanır. Sayaç, 16 bitlik değer maksimum değerine her ulaştığında bir kesme üretecek şekilde yapılandırılmıştır — etkin bir şekilde bir taşma kesmesi oluşturur. Kesme hizmet rutini, taşma oluşumlarının sayısını sayar.

32 bitlik değer, taşma oluşumlarının sayısı döndürülen 32 bitlik değerin en anlamlı iki baytı olarak ve mevcut 16 bitlik sayaç değeri 32 bitlik değerin en az anlamlı iki baytı olarak kullanılarak oluşturulur. Kesme hizmet rutini için sözde kod Liste 12.10'da gösterilmektedir.

```
void TimerOverflowInterruptHandler( void ) {     /* Sadece kesme sayısını
say. */     ulOverflowCount++;      /* Kesmeyi temizle. */
ClearTimerInterrupt(); }
```

Liste 12.10 Zamanlayıcı taşmalarını saymak için kullanılan 16 bitlik zamanlayıcı taşma kesme işleyicisi

Liste 12.11, çalışma zamanı istatistiklerinin toplanmasını etkinleştirmek için `FreeRTOSConfig.h`'ye eklenen satırları göstermektedir.

```
/* Çalışma zamanı istatistiklerinin toplanmasını etkinleştirmek için
   configGENERATE_RUN_TIME_STATS'ı 1 olarak ayarlayın. Bu yapıldığında,
   portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() ve portGET_RUN_TIME_COUNTER_VALUE()
   veya portALT_GET_RUN_TIME_COUNTER_VALUE(x) de tanımlanmalıdır. */
#define configGENERATE_RUN_TIME_STATS 1

/* portCONFIGURE_TIMER_FOR_RUN_TIME_STATS(), varsayımsal 16-bit
   zamanlayıcıyı kuran işlevi çağırmak üzere tanımlanır (işlevin
   uygulaması gösterilmemiştir). */
void vSetupTimerForRunTimeStats( void );
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()   vSetupTimerForRunTimeStats()

/* portALT_GET_RUN_TIME_COUNTER_VALUE(), parametresini geçerli çalışma
   zamanı sayaç/zaman değerine ayarlamak üzere tanımlanır. Döndürülen
   zaman değeri 32 bittir ve 16-bit zamanlayıcı taşma sayısını 32-bit
   bir sayının üst iki baytına kaydırarak, ardından sonucu geçerli
   16-bit sayaç değeri ile bit düzeyinde VEYA işlemine tabi tutarak
   oluşturulur. */
#define portALT_GET_RUN_TIME_COUNTER_VALUE( ulCountValue )              \
{                                                                       \
    extern volatile unsigned long ulOverflowCount;                      \
                                                                        \
    /* Değeri kullanılırken değişmemesi için saati sayaçtan             \
```

```
       ayırın. */                                                       \
    PauseTimer();                                                       \
                                                                        \
    /* Taşma sayısı, döndürülen 32-bit değerin en anlamlı              \
       iki baytına kaydırılır. */                                       \
    ulCountValue = ( ulOverflowCount << 16UL );                         \
                                                                        \
    /* Geçerli sayaç değeri, döndürülen 32-bit değerin en az           \
       anlamlı iki baytı olarak kullanılır. */                          \
    ulCountValue |= ( unsigned long ) ReadTimerCount();                 \
                                                                        \
    /* Saati sayaca yeniden bağlayın. */                                \
    ResumeTimer();                                                      \
}
```

Liste 12.11 Çalışma zamanı istatistiklerinin toplanmasını etkinleştirmek için FreeRTOSConfig.h'ye eklenen makrolar

Liste 12.12'de gösterilen görev, toplanan çalışma zamanı istatistiklerini her 5 saniyede bir yazdırır.

```
#define RUN_TIME_STATS_STRING_BUFFER_LENGTH    512

/* Netlik için, fflush() çağrıları bu kod listesinden çıkarılmıştır. */
static void prvStatsTask( void *pvParameters )
{
    TickType_t xLastExecutionTime;

    /* Biçimlendirilmiş çalışma zamanı istatistikleri metnini tutacak
       arabelleğin oldukça büyük olması gerekir. Bu nedenle, görev
       yığınında tahsis edilmemesini sağlamak için static olarak
       bildirilir. Bu, işlevi yeniden girişsiz (non re-entrant) yapar. */
    static signed char cStringBuffer[ RUN_TIME_STATS_STRING_BUFFER_LENGTH ];

    /* Görev her 5 saniyede bir çalışacaktır. */
    const TickType_t xBlockPeriod = pdMS_TO_TICKS( 5000 );

    /* xLastExecutionTime'ı geçerli zamana başlatın. Bu değişkenin
       açıkça yazılması gereken tek yer burasıdır. Sonrasında
       vTaskDelayUntil() API işlevi içinde dahili olarak güncellenir. */
    xLastExecutionTime = xTaskGetTickCount();

    /* Çoğu görevde olduğu gibi, bu görev sonsuz bir döngüde
       uygulanmıştır. */
    for( ;; )
    {
        /* Bu görevi tekrar çalıştırma zamanı gelene kadar bekleyin. */
        xTaskDelayUntil( &xLastExecutionTime, xBlockPeriod );

        /* Çalışma zamanı istatistiklerinden bir metin tablosu
           oluşturun. Bu, cStringBuffer dizisine sığmalıdır. */
```

```
        vTaskGetRunTimeStatistics( cStringBuffer,
RUN_TIME_STATS_STRING_BUFFER_LENGTH );

        /* Çalışma zamanı istatistikleri tablosu için sütun
           başlıklarını yazdırın. */
        printf( "\nTask\t\tAbs\t\t%%\n" );
        printf( "---------------------------------------------\n" );

        /* Çalışma zamanı istatistiklerini yazdırın. Veri tablosu
           birden fazla satır içerdiğinden, printf() doğrudan
           çağrılmak yerine vPrintMultipleLines() işlevi çağrılır.
           vPrintMultipleLines(), satır arabellemesinin beklendiği
           gibi çalışmasını sağlamak için her satırda ayrı ayrı
           printf() çağırır. */
        vPrintMultipleLines( cStringBuffer );
    }
}
```

Liste 12.12 Toplanan çalışma zamanı istatistiklerini yazdıran görev

12.6 İzleme Kanca Makroları

İzleme makroları, FreeRTOS kaynak kodu içindeki kilit noktalara yerleştirilmiş makrolardır. Varsayılan olarak, makrolar boştur ve bu nedenle herhangi bir kod üretmezler ve çalışma zamanı yükü yoktur. Varsayılan boş uygulamaları geçersiz kılarak, bir uygulama geliştiricisi şunları yapabilir:

• FreeRTOS kaynak dosyalarını değiştirmeden FreeRTOS'a kod eklemek.

• Hedef donanımda mevcut olan herhangi bir yolla ayrıntılı yürütme sıralama bilgileri çıktısı almak. İzleme makroları, FreeRTOS kaynak kodunda tam ve ayrıntılı bir çizgeleyici etkinlik izi ve profil çıkarma günlüğü oluşturmak için kullanılabilecek kadar çok yerde bulunur.

12.6.1 Kullanılabilir İzleme Kanca Makroları

Her makroyu burada ayrıntılı olarak açıklamak çok fazla yer kaplayacaktır. Aşağıdaki liste, bir uygulama geliştiricisi için en yararlı olduğu düşünülen makro alt kümesini detaylandırmaktadır.

Aşağıdaki listedeki açıklamaların birçoğu `pxCurrentTCB` adlı bir değişkene atıfta bulunur. `pxCurrentTCB`, Çalışıyor durumundaki görevin tanıtıcısını tutan ve `FreeRTOS/Source/tasks.c` kaynak dosyasından çağrılan herhangi bir makro tarafından kullanılabilen özel bir FreeRTOS değişkenidir.

En sık kullanılan izleme kanca makrolarının bir seçkisi:

`traceTASK_INCREMENT_TICK(xTickCount)` — Tick kesmesi sırasında, tick sayısı artırılmadan önce çağrılır. xTickCount parametresi yeni tick sayısı değerini makroya geçirir.

`traceTASK_SWITCHED_OUT()` — Çalıştırılacak yeni bir görev seçilmeden önce çağrılır. Bu noktada `pxCurrentTCB`, Çalışıyor durumundan ayrılmak üzere olan görevin tanıtıcısını içerir.

`traceTASK_SWITCHED_IN()` — Çalıştırılacak bir görev seçildikten sonra çağrılır. Bu noktada `pxCurrentTCB`, Çalışıyor durumuna girmek üzere olan görevin tanıtıcısını içerir.

`traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)` — Boş bir kuyruktan okuma girişiminin veya boş bir semafor ya da mutex'i 'alma' girişiminin ardından mevcut yürütülen görev Engellenmiş duruma girmeden hemen önce çağrılır. pxQueue parametresi hedef kuyruk veya semafor tanıtıcısını makroya geçirir.

`traceBLOCKING_ON_QUEUE_SEND(pxQueue)` — Dolu bir kuyruğa yazma girişiminin ardından mevcut yürütülen görev Engellenmiş duruma girmeden hemen önce çağrılır. pxQueue parametresi hedef kuyruk tanıtıcısını makroya geçirir.

`traceQUEUE_SEND(pxQueue)` — `xQueueSend()`, `xQueueSendToFront()`, `xQueueSendToBack()` veya semafor 'verme' işlevlerinden herhangi birinin içinden, kuyruk gönderimi veya semafor 'verme' başarılı olduğunda çağrılır.

`traceQUEUE_SEND_FAILED(pxQueue)` — `xQueueSend()`, `xQueueSendToFront()`, `xQueueSendToBack()` veya semafor 'verme' işlevlerinden herhangi birinin içinden, kuyruk gönderimi veya semafor 'verme' işlemi başarısız olduğunda çağrılır. Kuyruk doluysa ve belirtilen herhangi bir engelleme süresi boyunca dolu kalırsa bir kuyruk gönderimi veya semafor 'verme' başarısız olacaktır.

`traceQUEUE_RECEIVE(pxQueue)` — `xQueueReceive()` veya semafor 'alma' işlevlerinden herhangi birinin içinden, kuyruk alımı veya semafor 'alma' başarılı olduğunda çağrılır.

`traceQUEUE_RECEIVE_FAILED(pxQueue)` — `xQueueReceive()` veya semafor 'alma' işlevlerinden herhangi birinin içinden, kuyruk veya semafor alma işlemi başarısız olduğunda çağrılır. Kuyruk veya semafor boşsa ve belirtilen herhangi bir engelleme süresi boyunca boş kalırsa bir kuyruk alımı veya semafor 'alma' işlemi başarısız olacaktır.

`traceQUEUE_SEND_FROM_ISR(pxQueue)` — `xQueueSendFromISR()` içinden gönderme işlemi başarılı olduğunda çağrılır.

`traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)` — `xQueueSendFromISR()` içinden gönderme işlemi başarısız olduğunda çağrılır. Kuyruk zaten doluysa gönderme işlemi başarısız olacaktır.

`traceQUEUE_RECEIVE_FROM_ISR(pxQueue)` — `xQueueReceiveFromISR()` içinden alma işlemi başarılı olduğunda çağrılır.

`traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)` — `xQueueReceiveFromISR()` içinden kuyruk zaten boş olduğu için alma işlemi başarısız olduğunda çağrılır.

`traceTASK_DELAY_UNTIL(xTimeToWake)` — `xTaskDelayUntil()` içinden çağıran görev Engellenmiş duruma girmeden hemen önce çağrılır.

`traceTASK_DELAY()` — `vTaskDelay()` içinden çağıran görev Engellenmiş duruma girmeden hemen önce çağrılır.

12.6.2 İzleme Kanca Makrolarını Tanımlama

Her izleme makrosunun varsayılan bir boş tanımı vardır. Varsayılan tanım, `FreeRTOSConfig.h` içinde yeni bir makro tanımı sağlanarak geçersiz kılınabilir.

İzleme makrosu tanımları uzun veya karmaşık hale gelirse, bunlar daha sonra `FreeRTOSConfig.h`'den dahil edilen yeni bir başlık dosyasında uygulanabilir.

Yazılım mühendisliği en iyi uygulamalarına uygun olarak, FreeRTOS katı bir veri gizleme politikası sürdürür. İzleme makroları kullanıcı kodunun FreeRTOS kaynak dosyalarına eklenmesine izin verir, bu nedenle izleme makrolarına görünür veri türleri uygulama koduna görünür olanlardan farklı olacaktır:

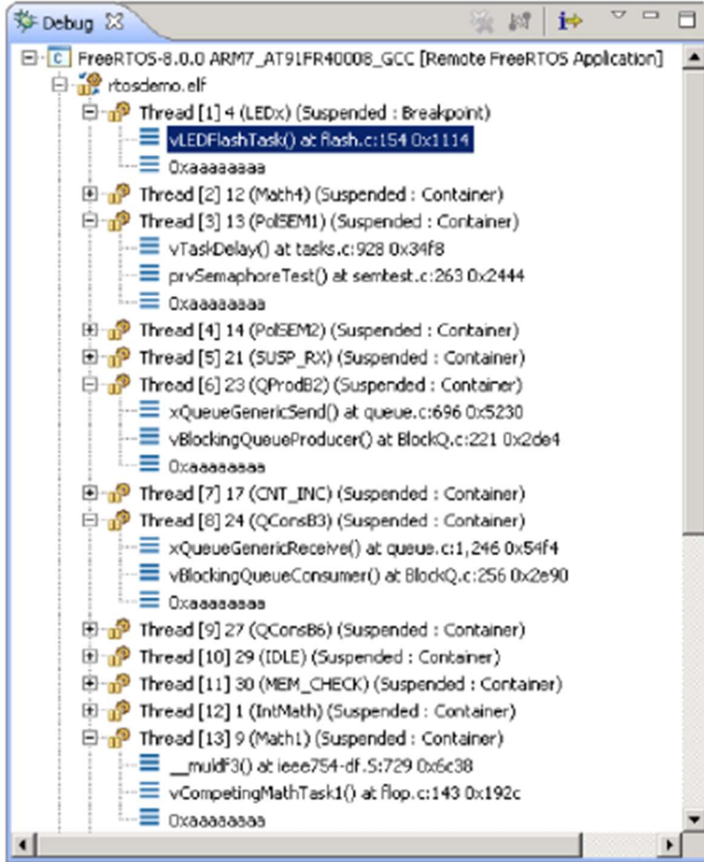
• `FreeRTOS/Source/tasks.c` kaynak dosyası içinde, bir görev tanıtıcısı bir görevi tanımlayan veri yapısına (görevin Görev Kontrol Bloğu veya TCB) bir işaretçidir. `FreeRTOS/Source/tasks.c` kaynak dosyası dışında bir görev tanıtıcısı void'e bir işaretçidir.

• `FreeRTOS/Source/queue.c` kaynak dosyası içinde, bir kuyruk tanıtıcısı bir kuyruğu tanımlayan veri yapısına bir işaretçidir. `FreeRTOS/Source/queue.c` kaynak dosyası dışında bir kuyruk tanıtıcısı void'e bir işaretçidir.

Normalde özel olan FreeRTOS veri yapısına doğrudan bir izleme makrosu tarafından erişiliyorsa son derece dikkatli olunmalıdır, çünkü özel veri yapıları FreeRTOS sürümleri arasında değişebilir.

12.6.3 FreeRTOS Uyumlu Hata Ayıklayıcı Eklentileri

Bir miktar FreeRTOS farkındalığı sağlayan eklentiler aşağıdaki IDE'ler için mevcuttur. Bu liste kapsamlı olmayabilir:



- Eclipse (StateViewer)
- Eclipse (ThreadSpy)
- IAR • ARM DS-5
- Atollic TrueStudio
- Microchip MPLAB
- iSYSTEM WinIDEA
- STM32CubeIDE

Bölüm 13

Sorun Giderme (Troubleshooting)

13.1 Bölüm Girişi (Introduction) ve Kapsamı (Scope)

Bu bölüm, FreeRTOS'ta yeni olan kullanıcılar tarafından karşılaşılan en yaygın sorunları vurgulamaktadır (highlights). İlk olarak (First), yıllar (years) boyunca (over) en sık (frequent) destek (support) talebi kaynağı (source) olduğu kanıtlanan (proven) üç konuya (issues) odaklanmaktadır (focuses): yanlış kesme önceliği ataması (incorrect interrupt priority assignment), yığın taşması (stack overflow) ve `printf()` işlevinin uygunsuz (inappropriate) kullanımı. Ardından (Then) SSS (FAQ - Sıkça Sorulan Sorular) tarzında, diğer yaygın hatalara, olası nedenlerine ve çözümlerine kısaca (briefly) değinir (touches on).

`configASSERT()` kullanmak, en yaygın (common) hata kaynaklarının çoğunu (many of) anında yakalayarak (trapping) ve tanımlayarak (identifying) üretkenliği (productivity) artırır (improves). Bir FreeRTOS uygulaması geliştirirken (developing) veya hata ayıklarken (debugging) `configASSERT()` tanımlanmış (defined) olması şiddetle (strongly) tavsiye edilir (advised). `configASSERT()` Bölüm 12.2'de açıklanmıştır (described).

13.2 Kesme Öncelikleri (Interrupt Priorities)

Not: Bu, destek taleplerinin (support requests) bir numaralı nedenidir (cause) ve çoğu bağlantı noktasında (ports) `configASSERT()` tanımlamak hatayı hemen yakalayacaktır (trap)!

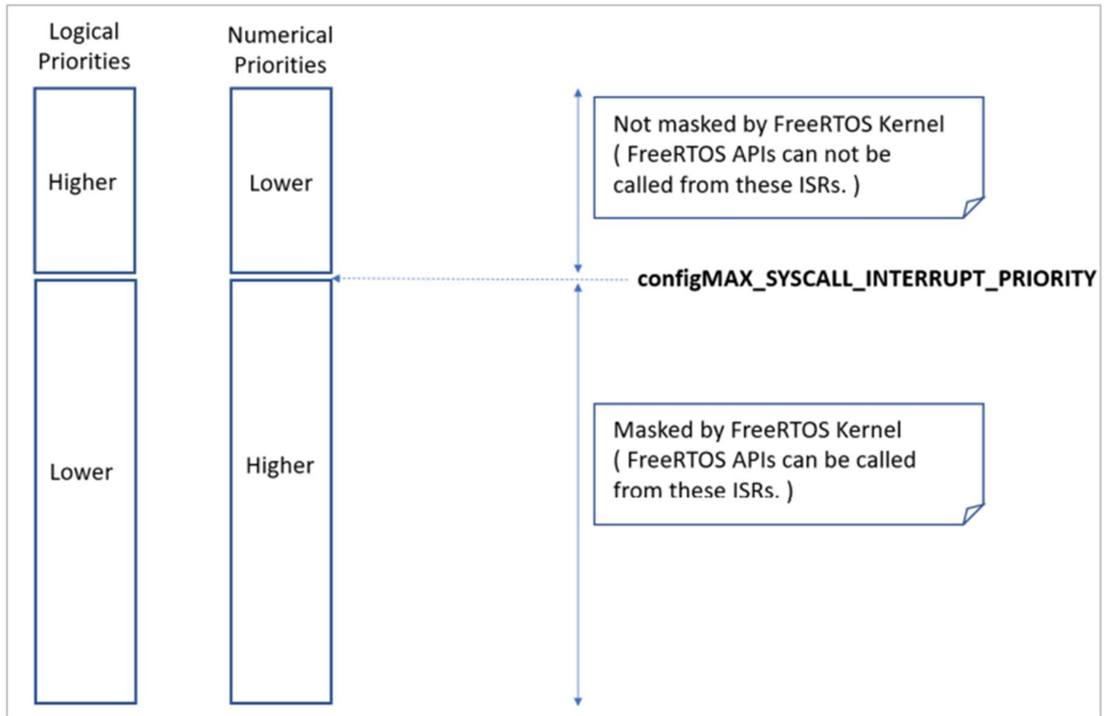
Kullanımda olan (in use) FreeRTOS bağlantı noktası kesme iç içe geçmesini (interrupt nesting) destekliyorsa (supports) ve bir kesme için hizmet rutini (service routine) FreeRTOS API'sinden yararlanıyorsa (makes use of), Bölüm 7.8, Kesme İç İçe Geçmesi'nde açıklandığı gibi kesmenin önceliğinin `configMAX_SYSCALL_INTERRUPT_PRIORITY` değerine veya daha altına (below) ayarlanması esastır (essential). Bunun yapılmaması (Failure to do this), etkisiz (ineffective) kritik bölümlerle (critical sections) sonuçlanacak (result in), bu da aralıklı (intermittent) arızalara (failures) yol açacaktır (result in turn).

FreeRTOS aşağıdaki (where) durumlarda (processor) çalıştırılıyorsa (running) özellikle (particular) dikkat edin:

- **Kesme öncelikleri varsayılan olarak (default to) mümkün olan en yüksek (highest possible) önceliğe sahiptir,** ki bu (which is the case)

bazı (some) ARM Cortex işlemcilerinde ve muhtemelen diğerlerinde (possibly others) böyledir. Bu tür (such) işlemcilerde, FreeRTOS API'sini kullanan (uses) bir kesmenin önceliği (priority) ilklendirilmemiş (uninitialized) olarak bırakılamaz (cannot be left).

- **Sayısal olarak (Numerically) yüksek öncelik numaraları mantıksal (logically) olarak düşük kesme önceliklerini temsil eder (represent),** bu da mantığa aykırı (counterintuitive) görünebilir (seem) ve bu nedenle (therefore) kafa karışıklığına (confusion) neden olabilir (cause). Yine bu durum ARM Cortex işlemcilerde ve muhtemelen diğerlerinde geçerlidir (case).
  Örneğin (For example), böyle bir işlemcide öncelik (priority) 5'te yürütülen (executing) bir kesmenin (interrupt) kendisi (itself) önceliği 4 olan bir kesme (interrupt) tarafından kesilebilir (interrupted). Bu nedenle (Therefore), `configMAX_SYSCALL_INTERRUPT_PRIORITY` 5 olarak ayarlanırsa, FreeRTOS API'sini kullanan herhangi bir kesmeye yalnızca sayısal olarak (numerically) 5'ten yüksek veya (or) ona eşit (equal) bir öncelik atanabilir (assigned). Bu durumda (In that case), 5 veya 6 kesme öncelikleri geçerli (valid) olacaktır, ancak 3 kesme önceliği kesinlikle (definitely) geçersizdir (invalid).



- **Farklı kütüphane (library) uygulamaları (implementations) bir kesmenin önceliğinin farklı bir şekilde belirtilmesini (specified) bekler.** Yine, bu özellikle (particularly) kesme önceliklerinin donanım yazmaçlarına (hardware registers) yazılmadan (written) önce (before) bit kaydırmasına (bit shifted) tabi tutulduğu (where) ARM Cortex işlemcileri hedefleyen (target) kütüphanelerle ilgilidir (relevant to). Bazı kütüphaneler (libraries) bit kaydırmayı (bit shift) kendileri (themselves) gerçekleştirirken, diğerleri (others) öncelik (priority) kütüphane işlevine (library function)

geçirilmeden (passed into) önce bit kaydırmanın (bit shift) gerçekleştirilmesini (performed) bekler.

- **Aynı mimarinin (architecture) farklı uygulamaları (implementations) farklı sayıda kesme öncelik biti uygular.** Örneğin, bir üreticinin (manufacturer) Cortex-M işlemcisi 3 öncelik biti uygulayabilirken (implement), başka bir üreticinin Cortex-M işlemcisi 4 öncelik biti uygulayabilir.
- **Bir kesmenin önceliğini (priority) tanımlayan (define) bitler (bits),** ön (pre-emption) önceliği tanımlayan bitler (bits) ile alt (sub) önceliği tanımlayan bitler (bits) arasında (between) bölünebilir (split). Alt önceliklerin (sub-priorities) kullanılmaması için (so that) tüm bitlerin (all the bits) bir öncelik (pre-emption priority) belirlemeye (specifying) atandığından (assigned) emin olun (Ensure).

Bazı FreeRTOS bağlantı noktalarında (ports), `configMAX_SYSCALL_INTERRUPT_PRIORITY` alternatif (alternative) olarak `configMAX_API_CALL_INTERRUPT_PRIORITY` adına (name) sahiptir.

13.3 Yığın Taşması (Stack Overflow)

Yığın taşması, en yaygın (common) ikinci destek (support) talebi kaynağıdır. FreeRTOS, yığınla (stack) ilgili (related) sorunları (issues) yakalamaya (trapping) ve hatalarını ayıklamaya (debugging) yardımcı olacak (assist) çeşitli (several) özellikler (features) sunar[28].

(28): Bu özellikler FreeRTOS Windows bağlantı noktasında mevcut değildir.

13.3.1 uxTaskGetStackHighWaterMark() API İşlevi

Her görev (task) kendi yığınını (stack) korur (maintains) ve bunun toplam boyutu (total size) görev oluşturulduğunda (created) belirlenir (specified).

`uxTaskGetStackHighWaterMark()` işlevi, bir görevin (task) kendisine ayrılan (allocated) yığın alanını (stack space) taşırmaya (overflowing) ne kadar yaklaştığını (how close) sorgulamak (query) için kullanılır. Bu değere yığın 'yüksek su işareti' (high water mark) denir (called).

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

Liste 13.1 `uxTaskGetStackHighWaterMark()` API işlevi prototipi (prototype)

`uxTaskGetStackHighWaterMark()` parametreleri ve dönüş değeri:

- **xTask:** Yığın yüksek su işareti sorgulanan (queried) görevin (konu görev - subject task) tanıtıcısıdır (handle). Görevlere ait tanıtıcıların (handles) elde edilmesiyle ilgili bilgi için `xTaskCreate()` API işlevinin `pxCreatedTask` parametresine (parameter) bakın.

Bir görev (task), geçerli (valid) bir görev tanıtıcısı (handle) yerine `NULL` geçirerek (passing) kendi yığın (stack) yüksek su işaretini (high water mark) sorgulayabilir (query).

- **Döndürülen Değer:** Görev tarafından kullanılan (used) yığın (stack) miktarı (amount), görev yürütüldükçe (executes) ve kesmeler (interrupts) işlendikçe (processed) büyür (grows) ve küçülür (shrinks). `uxTaskGetStackHighWaterMark()`, görev (task) yürütülmeye (executing) başladığından beri (since) kullanılabilir (available) olan kalan (remaining) yığın (stack) alanının minimum (minimum) miktarını (amount) döndürür (returns). Bu, yığın (stack) kullanımı (usage) en yüksek (veya en derin) değerindeyken kullanılmayan (unused) yığın (stack) miktarıdır (amount). Yüksek su işareti (high water mark) sıfıra ne kadar yakınsa (closer), görev yığınını taşırmaya (overflowing) o kadar yaklaşmış demektir.

`uxTaskGetStackHighWaterMark2()` API'si, yalnızca (only) dönüş türünde (return type) farklılık (differs) gösteren `uxTaskGetStackHighWaterMark()` yerine kullanılabilir (used).

```
configSTACK_DEPTH_TYPE uxTaskGetStackHighWaterMark2( TaskHandle_t xTask );
```

Liste 13.2 `uxTaskGetStackHighWaterMark`2() API işlevi prototipi (prototype)

`configSTACK_DEPTH_TYPE` kullanımı (Using), uygulama yazarının (application writer) yığın (stack) derinliği (depth) için kullanılan türü (type) kontrol etmesini (control) sağlar.

13.3.2 Çalışma Zamanı (Run Time) Yığın (Stack) Kontrolü — Genel Bakış (Overview)

FreeRTOS, isteğe bağlı (optional) üç çalışma zamanı (run time) yığın denetim (checking) mekanizması içerir (includes). Bunlar, `FreeRTOSConfig.h` içindeki `configCHECK_FOR_STACK_OVERFLOW` derleme (compile) zamanı (time) yapılandırma sabiti (configuration constant) tarafından kontrol edilir (controlled). Her iki yöntem (methods) de bir bağlam (context) anahtarını (switch) gerçekleştirme (perform) süresini (time) artırır.

Yığın taşması (stack overflow) kancası (veya yığın taşması geri çağırması - callback), çekirdek bir yığın taşması tespit (detects) ettiğinde (when) çekirdek (kernel) tarafından çağrılan (called) bir işlevdir (function). Bir yığın taşması (stack overflow) kanca işlevi (hook function) kullanmak (use) için:

1. Aşağıdaki alt bölümlerde (sub-sections) açıklandığı (described) gibi, `FreeRTOSConfig.h` dosyasında `configCHECK_FOR_STACK_OVERFLOW` değerini 1, 2 veya 3 olarak ayarlayın.
2. Liste 13.3'te gösterilen tam (exact) işlev (function) adını ve prototipini kullanarak (using) kanca işlevinin uygulamasını (implementation) sağlayın (Provide).

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask, signed char *pcTaskName
);
```

Liste 13.3 Yığın taşması kancası (stack overflow hook) işlevi prototipi (prototype)

Yığın taşması kanca işlevi, yığın hatalarını yakalamayı ve hata ayıklamayı kolaylaştırmak için sağlanır, ancak bir yığın taşması gerçekleştiğinde kurtarmanın gerçek bir yolu yoktur. İşlevin parametreleri, yığınını taşıran görevin tanıtıcısını (handle) ve adını kanca işlevine iletir.

Yığın taşması kanca işlevi bir kesme bağlamından çağrılır.

Bazı mikrodenetleyiciler hatalı bir bellek erişimi algıladığında bir hata istisnası (fault exception) üretir ve çekirdek yığın taşması kanca işlevini çağırma fırsatı bulmadan önce bir hatanın tetiklenmesi mümkündür.

13.3.3 Çalışma Zamanı Yığın (Stack) Kontrolü — Yöntem (Method) 1

`configCHECK_FOR_STACK_OVERFLOW` 1 olarak ayarlandığında 1. yöntem (Method 1) seçilir (selected).

Bir görevin (task) tüm yürütme (execution) bağlamı (context), her dışarı (out) aktarıldığında (swapped) kendi yığınına (stack) kaydedilir (saved). Yığın (stack) kullanımının en yüksek noktasına (peak) ulaştığı (reaches) zamanın (time) bu zaman olması muhtemeldir (likely). `configCHECK_FOR_STACK_OVERFLOW` 1 olarak ayarlandığında, çekirdek (kernel) bağlam (context) kaydedildikten (saved) sonra yığın işaretçisinin (stack pointer) geçerli (valid) yığın alanında (stack space) kaldığını kontrol eder (checks). Yığın işaretçisinin geçerli (valid) aralığının (range) dışında (outside) olduğu bulunursa yığın taşması kancası (stack overflow hook) çağrılır (called).

Yöntem 1'in yürütülmesi (execute) hızlıdır, ancak bağlam anahtarları (context switches) arasında meydana (occur) gelen yığın taşmalarını (stack overflows) gözden kaçırabilir (miss).

13.3.4 Çalışma Zamanı Yığın (Stack) Kontrolü — Yöntem (Method) 2

Yöntem 2, yöntem 1 için zaten açıklanmış (described) olanlara (those) ek denetimler (additional checks) gerçekleştirir (performs). `configCHECK_FOR_STACK_OVERFLOW` 2 olarak ayarlandığında seçilir (selected).

Bir görev oluşturulduğunda (created), yığını (stack) bilinen bir desenle (known pattern) doldurulur (filled). Yöntem 2, bu desenin üzerine yazılmadığını (not been overwritten) doğrulamak (verify) için görev yığın alanının (stack space) son (last) geçerli 20 baytını (bytes) test eder (tests). 20 bayttan herhangi biri beklenen değerlerinden (expected values) değişmişse (changed) yığın taşması (stack overflow) kanca işlevi çağrılır.

Yöntem 2'nin yürütülmesi (execute) yöntem 1 kadar (as) hızlı değildir (not as quick), ancak (but) yine de nispeten (relatively) hızlıdır, çünkü yalnızca (only) 20 bayt test (tested) edilir. Büyük olasılıkla (Most likely), tüm yığın taşmalarını (stack overflows) yakalayacaktır (catch); ancak, bazı taşmaların gözden kaçırılması (missed) olasıdır (ama çok düşük bir ihtimaldir - highly improbable).

13.3.4 Çalışma Zamanı Yığın (Stack) Kontrolü — Yöntem (Method) 3

`configCHECK_FOR_STACK_OVERFLOW` 3 olarak ayarlandığında 3. yöntem seçilir (selected).

Bu yöntem yalnızca seçili (selected) bağlantı noktaları (ports) için mevcuttur (available). Kullanılabildiğinde (When available), bu yöntem ISR yığın (stack) denetimini (checking) etkinleştirir. Bir ISR yığın taşması tespit (detected) edildiğinde, bir iddia (assert) tetiklenir (triggered). Yığın taşması kanca işlevinin (stack overflow hook function) bu durumda (case) çağrılmadığını (not called), çünkü ISR yığınına (stack) değil, bir görev (task) yığınına özgü (specific) olduğuna (because it is) dikkat edin (Note).

13.4 printf() ve sprintf() Kullanımı

`printf()` ile günlükleme yaygın bir hata kaynağıdır ve bunun farkında olmadan uygulama geliştiricilerinin hata ayıklamaya yardımcı olmak için daha fazla `printf()` çağrısı eklemeleri ve böylece sorunu ağırlaştırmaları yaygındır.

Birçok çapraz derleyici satıcısı, küçük gömülü sistemlerde kullanıma uygun bir `printf()` uygulaması sağlayacaktır. Bu durumda bile, uygulama iş parçacığı güvenli olmayabilir, muhtemelen bir kesme hizmet rutini içinde kullanıma uygun olmayacaktır ve çıktının yönlendirildiği yere bağlı olarak yürütülmesi görece uzun sürebilir.

Özellikle küçük gömülü sistemler için tasarlanmış bir `printf()` uygulaması mevcut değilse ve bunun yerine genel bir `printf()` uygulaması kullanılıyorsa özel dikkat gösterilmelidir, çünkü:

- Yalnızca bir `printf()` veya `sprintf()` çağrısı dahil etmek, uygulamanın yürütülebilir dosyasının boyutunu büyük ölçüde artırabilir.
- `printf()` ve `sprintf()` `malloc()` çağırabilir, bu da `heap_3` dışında bir bellek ayırma şeması kullanılıyorsa geçersiz olabilir. Daha fazla bilgi için Bölüm 3.2, Örnek Bellek Ayırma Şemaları bölümüne bakın.
- `printf()` ve `sprintf()` normalde gerekli olandan çok daha büyük bir yığın gerektirebilir.

13.4.1 Printf-stdarg.c

FreeRTOS gösterim projelerinin birçoğu, standart kütüphane sürümü yerine kullanılabilecek minimal ve yığın açısından verimli bir `sprintf()` uygulaması sağlayan `printf-stdarg.c` adlı bir dosya kullanır. Çoğu durumda bu, `sprintf()` ve ilgili işlevleri çağıran her göreve çok daha küçük bir yığın ayrılmasına olanak tanıyacaktır.

`printf-stdarg.c` ayrıca `printf()` çıktısını karakter karakter bir bağlantı noktasına yönlendirmek için bir mekanizma sağlar; bu yavaş olmakla birlikte yığın kullanımının daha da azaltılmasını sağlar.

FreeRTOS indirmesine dahil edilen tüm `printf-stdarg.c` kopyalarının `snprintf()` uygulamadığını unutmayın. `snprintf()` uygulamayan kopyalar, doğrudan `sprintf()` ile eşleştirdikleri için arabellek boyutu parametresini yok sayarlar.

`printf-stdarg.c` açık kaynaklıdır, ancak üçüncü bir tarafa aittir ve bu nedenle FreeRTOS'tan ayrı olarak lisanslanmıştır. Lisans koşulları kaynak dosyanın üst kısmında yer almaktadır.

13.5 Diğer Yaygın Hata Kaynakları

13.5.1 Belirti: Bir demo'ya basit bir görev eklemek demo'nun çökmesine neden oluyor

Bir görev oluşturmak, yığın belleğinden (heap) bellek alınmasını gerektirir. Demo uygulama projelerinin birçoğu, yığın belleğini yalnızca demo görevlerini oluşturmaya yetecek kadar boyutlandırır — dolayısıyla görevler oluşturulduktan sonra, ek görevler, kuyruklar, olay grupları veya semaforlar eklemek için yeterli yığın belleği kalmayacaktır.

Boşta görevi ve muhtemelen RTOS arka plan görevi, `vTaskStartScheduler()` çağrıldığında otomatik olarak oluşturulur. `vTaskStartScheduler()` yalnızca bu görevlerin oluşturulması için yeterli yığın belleği kalmamışsa geri dönecektir. `vTaskStartScheduler()` çağrısından sonra bir boş döngü [`for(;;)`;] eklemek bu hatanın ayıklanmasını kolaylaştırabilir.

Daha fazla görev ekleyebilmek için ya yığın belleği boyutunu artırmanız ya da mevcut demo görevlerinden bazılarını kaldırmanız gerekir. Yığın belleği boyutundaki artış her zaman mevcut RAM miktarıyla sınırlı olacaktır. Daha fazla bilgi için Bölüm 3.2, Örnek Bellek Ayırma Şemaları bölümüne bakın.

13.5.2 Belirti: Bir kesme içinde API işlevi kullanmak uygulamanın çökmesine neden oluyor

API işlev adı '...FromISR()' ile bitmediği sürece kesme hizmet rutinleri içinde API işlevleri kullanmayın. Özellikle, kesme güvenli makrolar kullanmadan bir kesme içinde kritik bölüm oluşturmayın. Daha fazla bilgi için Bölüm 7.2, Bir KHR'den FreeRTOS API Kullanımı bölümüne bakın.

Kesme iç içe geçmesini destekleyen FreeRTOS bağlantı noktalarında, `configMAX_SYSCALL_INTERRUPT_PRIORITY` üzerinde bir kesme önceliği atanmış herhangi bir kesmede API işlevlerini kullanmayın. Daha fazla bilgi için Bölüm 7.8, Kesme İç İçe Geçmesi bölümüne bakın.

13.5.3 Belirti: Bazen uygulama bir kesme hizmet rutini içinde çöküyor

Kontrol edilecek ilk şey, kesmenin bir yığın taşmasına neden olup olmadığıdır. Bazı bağlantı noktaları yalnızca görevlerde yığın taşmasını kontrol eder, kesmelerde değil.

Kesmelerin tanımlanma ve kullanılma şekli bağlantı noktaları ve derleyiciler arasında farklılık gösterir. Bu nedenle, kontrol edilecek ikinci şey, kesme hizmet rutininde kullanılan söz dizimi, makrolar ve çağırma kurallarının, kullanılan bağlantı noktası için sağlanan belgeler sayfasında açıklandığı gibi ve bağlantı noktasıyla birlikte sağlanan demo uygulamasında gösterildiği gibi olmasıdır.

Uygulama, mantıksal olarak yüksek öncelikleri temsil etmek için sayısal olarak düşük öncelik numaraları kullanan bir işlemcide çalışıyorsa, sezgisel olmayan göründüğü için her kesmeye atanan önceliğin bunu dikkate aldığından emin olun. Daha fazla bilgi için Bölüm 7.8 ve Bölüm 13.2 bölümlerine bakın.

13.5.4 Belirti: İlk görevi başlatmaya çalışırken çizgeleyici çöküyor

FreeRTOS kesme işleyicilerinin yüklendiğinden emin olun. Bilgi için kullanılan FreeRTOS bağlantı noktasının belgeler sayfasına ve bir örnek için bağlantı noktasıyla birlikte sağlanan demo uygulamasına başvurun.

Bazı işlemcilerin, çizgeleyici başlatılabilmeden önce ayrıcalıklı modda olması gerekir. Bunu başarmanın en kolay yolu, main() çağrılmadan önce C başlangıç kodunda işlemciyi ayrıcalıklı moda yerleştirmektir.

13.5.5 Belirti: Kesmeler beklenmedik şekilde devre dışı bırakılıyor veya kritik bölümler düzgün iç içe geçmiyor

Çizgeleyici başlatılmadan önce bir FreeRTOS API işlevi çağrılırsa, kesmeler kasıtlı olarak devre dışı bırakılacak ve ilk görev yürütülmeye başlayana kadar yeniden etkinleştirilmeyecektir. Bu, çizgeleyici başlatılmadan önce ve çizgeleyici tutarsız bir durumda olabilirken, sistem başlatma sırasında FreeRTOS API işlevlerini kullanmaya çalışan kesmelerin neden olduğu çökmelerden sistemi korumak için yapılır.

`taskENTER_CRITICAL()` ve `taskEXIT_CRITICAL()` çağrıları dışında herhangi bir yöntemle mikrodenetleyici kesme etkinleştirme bitlerini veya öncelik bayraklarını değiştirmeyin. Bu makrolar, kesmelerin yalnızca çağrı iç içe geçmesi tamamen sıfıra çözüldüğünde yeniden etkinleştirilmesini sağlamak için çağrı iç içe geçme derinliklerinin sayısını tutar. Bazı kütüphane işlevlerinin kendilerinin kesmeleri etkinleştirebileceğini ve devre dışı bırakabileceğini unutmayın.

13.5.6 Belirti: Çizgeleyici başlamadan önce bile uygulama çöküyor

Potansiyel olarak bağlam anahtarına neden olabilecek bir kesme hizmet rutininin, çizgeleyici başlatılmadan önce yürütülmesine izin verilmemelidir. Aynı durum, bir kuyruk veya semafor gibi bir FreeRTOS nesnesine göndermeye veya nesneden almaya çalışan herhangi bir kesme hizmet rutini için de geçerlidir. Çizgeleyici başlayana kadar bağlam anahtarı gerçekleşemez.

Birçok API işlevi, çizgeleyici başlatılana kadar çağrılamaz. API kullanımını, `vTaskStartScheduler()` çağrıldıktan sonra bu nesnelerin kullanımı yerine görevler, kuyruklar ve semaforlar gibi nesnelerin oluşturulmasıyla sınırlamak en iyisidir.

13.5.7 Belirti: Çizgeleyici askıya alınmışken veya kritik bölüm içinde API işlevi çağırmak uygulamanın çökmesine neden oluyor

Çizgeleyici `vTaskSuspendAll()` çağrılarak askıya alınır ve `xTaskResumeAll()` çağrılarak devam ettirilir. Kritik bölüme `taskENTER_CRITICAL()` çağrılarak girilir ve `taskEXIT_CRITICAL()` çağrılarak çıkılır.

Çizgeleyici askıya alınmışken veya kritik bölüm içindeyken API işlevlerini çağırmayın.

13.6 Ek Hata Ayıklama Adımları

Yukarıda açıklanan yaygın nedenler kapsamında olmayan bir sorunla karşılaşırsanız, aşağıdaki hata ayıklama adımlarından bazılarını deneyebilirsiniz:

• Uygulamanızın FreeRTOSConfig dosyasında `configASSERT()` tanımlayın, malloc başarısız denetimini ve yığın taşması denetimini etkinleştirin.

• FreeRTOS API'lerinin dönüş değerlerini kontrol ederek başarılı olduklarından emin olun.

• `configUSE_TIME_SLICING` ve `configUSE_PREEMPTION` gibi çizgeleyici ile ilgili yapılandırmanın, uygulama gereksinimlerine göre doğru ayarlandığını kontrol edin.

Cortex-M mikrodenetleyicilerinde hard fault hatalarını ayıklama hakkında ayrıntılı bilgi FreeRTOS belgelerinde mevcuttur.